

An Analysis of a Virtually Synchronous Protocol

Dan Arnon and Navindra Sharma
EMC Corporation

March 10, 2015

Abstract

Enterprise-scale systems such as those used for cloud computing require a scalable and highly available infrastructure. One crucial ingredient of such an infrastructure is the ability to replicate data coherently among a group of cooperating processes in the presence of process failures and group membership changes. The last few decades have seen prolific research into efficient protocols for such data replication. One family of such protocols are the virtually synchronous protocols. Virtually synchronous protocols achieve their efficiency by limiting their synchronicity guarantee to messages that bear a causal relationship to each other. Such protocols have found wide-ranging commercial uses over the years. One protocol in particular, the **CBCAST** protocol developed by Birman, Schiper and Stephenson in 1991 and used in their ISIS platform was particularly promising due to its unique no-wait properties, but has suffered from seemingly intractable race conditions. In this paper we describe a corrected version of this protocol and prove its formal properties.

Contents

1	Introduction	4
2	The Underlying Computational Model	6
2.1	Introduction	6
2.2	Model Overview	7
2.3	The Formal Model	9
2.3.1	Ingredients of the model	9
2.3.2	The PROTOCOL interface	11
2.3.3	The APP interface	13
2.3.4	Model axioms and histories	14
2.4	Event Order and Time	17
2.4.1	König’s Lemma	18
2.4.2	Stratifying events by view	19
2.4.3	Creating an event timeline	21
2.5	Understanding Faults	24
2.5.1	Conforming models and conforming histories	26
2.5.2	Fault equivalence	27
2.5.3	The Vacuum Loop and vacuum closure	28
2.5.4	Transactional histories and the Fault Theorem	32
3	The CBCAST Algorithm	38
3.1	Introduction	38
3.2	Terminology	38
3.2.1	Messages and delivery	38
3.2.2	Views, installations and view gaps	38
3.2.3	Instability and forwarding	39
3.2.4	Vector time	40
3.2.5	Cluster Initialization and original processes	40
3.3	Outline of the Algorithm	40
3.4	Variable and Function Definitions	41
3.4.1	Global variables - the state of a process	41
3.4.2	Packet and notification types	43
3.4.3	Message metadata	43
3.5	Detailed CBCAST Algorithm Pseudo Code	44
4	Basic Properties Of The CBCAST Algorithm	57
4.1	Some CBCAST invariants	57
4.2	Side Effects of CBCAST Triggers	72
4.2.1	Side effects of message broadcast request events	72
4.2.2	Side effects of view change notifications	72
4.2.3	Side effects of message and acknowledgement packet receipts	73
4.2.4	Side effects of ghost and flush packet receipts	74
4.2.5	Side effects of donation and co-donation packet receipts	74
4.3	CBCAST is Vacuum Convergent	75

5	The History Reduction Mapping	76
5.1	Introduction	76
5.2	Preliminaries	78
5.2.1	Transactions	78
5.2.2	A clean event order	78
5.2.3	Donation and co-donation sub-transactions	79
5.3	The Label Space	88
5.3.1	The constellation coordinate set	89
5.3.2	The sub-transaction coordinate set	89
5.3.3	The adjustment coordinate set	89
5.3.4	The side-effect coordinate set	90
5.3.5	Labeling the events in H	90
5.3.6	Some labeling examples	92
5.4	Defining The History Reduction Mapping	95
5.4.1	Preliminaries	95
5.4.2	Some notation	96
5.4.3	Constructing the new queuing events	97
5.4.4	Constructing the channels \overrightarrow{PQ} for non- G processes P and Q	99
5.4.5	Constructing the channels \overrightarrow{PG} and $\overrightarrow{P(-G)}$ for a non- G process P	99
5.4.6	Constructing the channels \overrightarrow{GP} and $\overrightarrow{(-G)P}$ for a non- G process P	100
5.4.7	Constructing the channels $(\pm G)(\pm G)$	101
5.4.8	Constructing the notification events	102
5.4.9	The partial order \prec^{H^r} and dropped items	103
5.4.10	Implementing the user application interface	104
5.5	Basic Properties Of H^r	106
6	The History Equivalence Theorem	115
6.1	Introduction	115
6.2	Proof plan and preliminaries	116
6.3	Side Effects in H^r	118
6.4	The inductive hypothesis	119
6.5	Technical lemmas for the History Equivalence Theorem proof	123
6.6	Proof of the History Equivalence Theorem	131
6.6.1	Notification constellations	132
6.6.2	Message broadcast request constellations	142
6.6.3	Message and acknowledgment packet constellations	144
6.6.4	Ghost and flush packet constellations	151
6.6.5	Donation and co-donation packet constellations	154
7	Causality and Progress With The CECAST Algorithm	163
7.1	The Goals of the Analysis	163
7.2	The Causal Order Property and The Progress Property	163
7.3	Causal Order And Progress In Reduced Histories	164
7.4	Stunting	166
7.5	The Central Lemma	167
7.6	Proving the Central Lemma	172

1 Introduction

Many modern computational tasks are performed by groups of processes that are physically distributed and are prone to failure. Such computations require efficient ways to reliably multicast messages within the group in the presence of group membership changes. These requirements are becoming increasingly common in data center systems such as storage systems and server clusters. Meeting these requirements while maintaining good performance is challenging. There is a need to keep all the members of the group in a synchronized state, whatever that may mean, and there is a need to avoid split brains and subtle race conditions that can occur when the group is reconfigured.

These requirements generally fall into three categories:

Message synchrony For the group to work coherently, at least some messages must be delivered at different group members in the same order.

Reconfiguration atomicity When the group membership changes, there must be some guarantee against split brains and all the members of the new group must reach some kind of agreement on a common initial state.

Progress guarantee There must be some guarantee that messages get delivered to the whole group.

Meeting such requirements can be costly. To reduce these costs different authors have proposed weaker requirements within these three categories that allow for more performant systems. The original papers describing the process group model [8, 4] envisioned incremental group reconfigurations, possibly limited to one member addition or removal at a time. This makes reconfiguration more expensive. More recent treatments of the subject assume general reconfigurations that occur in bulk. In [3] bulk reconfiguration is joined with a flexible framework of delivery guarantees that allows for the tailoring of these guarantees to the needs of specific applications. However this framework requires all message delivery to stop while the system is being reconfigured. The Rambo system [13] is designed more specifically for a distributed atomic memory, but it allows messages to flow while bulk reconfiguration is taking place. It also enjoys weaker synchrony guarantees that are tailored to the specific requirements of atomic memory.

One multicast paradigm that proved especially useful is Virtually Synchrony (see [4]). Virtual Synchrony achieves high efficiency albeit with some reduced availability (see [7]) by serializing messages only when they may be causally linked. Messages that are not causally linked may be delivered in different orders to different members of the process group. Virtually synchronous multicast protocols have been used for a long time in many commercial and non-commercial systems, for example Horus (see [5]) and Transis (see [1]). For a comprehensive specification of group multicast protocols and their properties, see [9].

One truly exceptional proposal for a virtually synchronous protocol was described in [6]. The **CBCAST** protocol that the authors describe in that paper not only promises virtual synchrony and reconfiguration atomicity without stopping message delivery, but also promises to work without any delivery guarantee - all message broadcasts are ship-and-pray. While this last promise is not explicitly elaborated on in the paper, it is what makes the **CBCAST** protocol exceptionally powerful. Unfortunately, **CBCAST** did not deliver on its promise. Its very complexity meant that race conditions could never be completely wrung out of it. The goal of this paper is to fix that protocol and provide

a rigorous proof of its properties.

Following [6], we describe the **CBCAST** algorithm within a formal model that contains processes, channels, packets, and an opaque group membership service. Channels connect pairs of processes, enabling them to send point-to-point packets (we reserve the term "messages" to the entities that are multicast by **CBCAST**). The group membership service (**GMS**) is assumed to be a primary-component **GMS** (as opposed to a partitionable one, see [9].) **GMS** provides ordered notifications of group membership changes. Each component of the model can fail independently. As a result, failure scenarios can get very complex.

We provide a rigorous proof of two essential properties of the **CBCAST** protocol, the Causal Order Property and the Progress Property. The Causal Order Property says that messages are delivered at each process in an order that respects the causality relationships between messages. The Progress Property says that if two processes in the group never halt, then any message broadcast by one is delivered at the other, provided that only a finite number of processes join the process group. Both of these properties are proved to hold under any pattern of component failure in the cluster.

The proof is divided into a number of parts, each of which is a separate investigation. The outline of the proof plan is as follows:

- The first part deals with the formal model only and is independent of **CBCAST**. We analyze the failure scenarios in the model (stop failures only) using an axiomatic approach, and show that under reasonable assumptions on the behavior of the **GMS** all partial failure cases are equivalent to either the failure-free case or to a simultaneous failure of all the components. In essence the group behaves like a single fault domain. This frees us to carry out the rest of the proof under the assumption that no failures ever occur. A critical ingredient in the analysis is the fact that the formal act of removing a process from the group by the **GMS** can mask the stop-failure of a process.
- In the second part we give a detailed description of the **CBCAST** protocol. The version described here is not the most general and certainly not the most efficient. Our goal here is to achieve maximum simplicity in order to facilitate a clear analysis. Some parts of the protocol that deal with the admission of processes into the group are new. Specifically the steps of state *donation* and *co-donation*.
- The third part deals with processes that are admitted to the group by **GMS** (as opposed to processes that are in the group from the start). We prove the rather surprising fact that a process admission event can be reduced to a process removal event. To do that we construct an explicit mapping from a history that contains an admission of a process G to a history that contains a removal of an "opposite" process $-G$. We show that the two histories carry the same computation and share the same progress and causality order properties. As a result we can get rid of any finite number of **GMS** admission events.
- The fourth part is the proof of the Causal Order Property and the Progress Property in the special case where no new processes are ever admitted to the group beyond the original members. A central idea in the proof is the concept of *effective routes*. The **CBCAST** protocol allows a message to be transmitted from source to target multiple times and through multiple routes. However only one of the routes is effective and all the other are redundant. Concentrating on the effective routes simplifies the analysis a great deal.

Our model makes a number of assumptions on the behavior of the group membership service. There are many different implementations of such a service in the literature. Such a service is sometimes referred to as a *reconfiguration* service. See for example [3, 13]. Virtually all such services are based on the Paxos protocol (see [11, 12]). Elsewhere we describe a detailed implementation and analysis of a compliant (primary-component) group membership service (see [2]).

2 The Underlying Computational Model

2.1 Introduction

In this section we create a detailed, axiomatic computational model in which the CBCAST protocol executes. The purpose of this model is to create a context that is rich and precise enough for us to carry two of the core arguments in this paper. First, that the failure model of a group computation with a group membership service and stop faults is equivalent to the failure model of a single process, where the only failure is an instantaneous failure of the whole system; Second, that any group computation that is based on CBCAST where a process G joins in the middle of the computation is identical to a computation carried by a similar group that includes the process G from the start. The first argument allows us to carry out our analysis of the properties of CBCAST without taking failures into consideration. The second allows us to carry the analysis under the assumption that processes never join the group during the computation.

The first argument is carried out in the current part. We use the interface points between different components in the model as a place to "shift the blame" from channel and GMS failures to process failures. Then we hide the process failures by subsuming them into the group membership service itself. To put this last shift in other words, if a process is officially removed by the group membership service and halts at the same time, then we can analyze the event as a pure group membership service event rather than as two separate events (a group membership service event and a component failure). To make this intuitive, process removal and process halting have to occur simultaneously. The notion of simultaneity requires some work since our model does not include a notion of time. We perform a careful analysis of the partial order that exists between events in the model to show that they can be laid along a timeline in such a way that desired events (such as process removal and process halting) occur at the same time.

To make blame shifting possible, we model the channels as including a queuing stage at the sending and receiving ends of the channel - in other words we add a send queue and a receive queue to each channel. A channel that fails to ship a packet to its destination can shift the blame to the sending process by claiming that the faulty packet never left the send queue. We add a similar queue for GMS notifications. This queuing construct is not as artificial as it may sound. Queuing is inherent to communications networks. The sender/channel and channel/receiver boundaries are inherently blurry.

The main technical difficulty in the analysis is keeping infinities from creeping in. If we try to shift the blame for an infinite number of failures to a single component of the model, we end up with the absurd conclusion that an infinite number of events happen in a finite period of time. It is for this reason that the analysis proceeds by dealing with each class of failures in one fell swoop rather than dealing with failures one at a time.

2.2 Model Overview

We assume a group of *processes* - the group having a set of initial members to which at various times new processes can be added, while processes that are already in the group can be removed. We do not care how the decision to add or remove processes is made. We assume that there is an opaque *group membership service* (**GMS** for short) that notifies all the current member processes of any addition or removal of a process. The notifications for each process are appended by **GMS** to a *GMS receive queue*. Eventually the process dequeues each notification in order and processes it. We do not assume that the membership service is reliable - notifications can stop arriving at an eligible process.

We assume that the processes can communicate with each other by exchanging packets on point-to-point, unidirectional *channels*. We assume that the packets in each channel arrive in the order they were sent, i.e. the channel between the source and target is FIFO. The channels are not assumed to be reliable. When a process P wants to send a packet to a process Q , it appends the packet to the *send queue* of the outbound channel between P and Q . Eventually an opaque driver dequeues the packet and sends it through the channel. When Q receives a packet from P , the packet is appended to the *receive queue* of the inbound channel by an opaque driver. Eventually the process dequeues each packet and processes it. To make the model more symmetrical we assume that there is a channel, called the *self-channel*, between each process and itself.

We assume that each process executes some code. This code is made up of a *protocol* **PROTOCOL** and an *application* **APP**. The application is an arbitrary execution thread that makes *message multicast requests* to communicate with other processes. Each message multicast request is appended to the *APP receive queue* of the process. Eventually the process dequeues each message and multicasts it.

APP does not directly specify the set of recipient processes of each multicast. Doing so is impossible since the roster of processes changes over time and messages are multicast asynchronously rather than immediately. Instead, **APP** must specify a *functional set* of recipients, and leaves it to **PROTOCOL** to decide who are the members of the functional set at the moment that it performs the multicast.

In general many functional sets can be specified for a multicast. For example if a natural order relation exists between all possible processes (e.g. an order on their identifiers) then the smallest member process, largest member process, or two smallest processes can be specified as functional sets. If the processes have distinguishing characteristics (e.g. colors) then the blue processes or ultraviolet processes can be specified.

In our analysis we do not assume any order or distinguishing characteristics. This leaves us with only one interesting functional set, namely the universal set of all member processes. A multicast to the universal set is called a *broadcast*. This is not really a limitation because functional sets can be specified by **APP** in the body of each message. Once the message is received by a process, the receiving process can decide on its own whether it belongs to the specified set. If it does not it can simply ignore the message.

PROTOCOL is a set of non-blocking callbacks that the process uses to process the packets, notifications and message broadcast requests that it dequeues from its various receive queues. We assume that the processing of each received *item* - packet, notification or message broadcast request - completes without any context switching, meaning that the items are processed one at a time; that items that arrive at the same queue are processed in the order at which they were queued; and that **APP** does

not progress while an item is processed.

We assume that each process has a *state*. The state can be thought of as the totality of values of all the variables that are managed by **PROTOCOL**. We assume that **APP** can read some of this state, but cannot change any of it directly. This does not mean that **APP** has no private state of its own, however we do not model it. We will see that at least in the case of **CBCAST**, and plausibly in the general case, **APP** is started from scratch at each new process and therefore we can treat it as stateless for the purpose of our analysis.

Since the state is only changed by **PROTOCOL**, the state of a process when it dequeues the next packet, notification or message broadcast request is identical to the state it had after it finished processing the previous packet, notification or message broadcast request. The initial group of processes all start their life with an identical initial state. Any process that is subsequently admitted into the group by **GMS** starts life with a state that is identical to the state of a *parent* which must be an existing group member. The parent is selected by **GMS**. To be more accurate, the new process starts life by dequeuing the **GMS** notification of its own joining. At that moment it has the same state that its parent has when it dequeues the same notification from **GMS**. This makes the act of dequeuing the notification similar to the execution of a `fork()` system call in UNIX.

There are a few types of *events* in our model. Each event occurs at a specific process and can be either a *queuing* event, also called a *side effect* event, or a *dequeuing event*, also called a *trigger* event.

A trigger event occurs when any item is dequeued from some receive queue. This includes packet, notification and message broadcast request dequeuing events. Such an event triggers processing, using the appropriate callback supplied by **PROTOCOL**.

A side effect event occurs when the process multicasts information by appending one or more identical packets to the send queues of outbound channels, destined to different processes. Such an event always occurs as a result of the execution of a callback, and is therefore a side effect of that execution. In other words, the side effects of a trigger are determined by the current state of the process and by the implementation of **PROTOCOL**. **APP** cannot initiate a queuing event directly, but only through the issue of an message broadcast request.

The various send queues of outbound channels; the receive queues of inbound channels; and the receive queues of notifications and message broadcast requests form an integral part of a process in our model. The inclusion of queuing as part of the model is in recognition of the fact that queuing is a fundamental property of any communication mechanism and is not an artifact of any particular implementation. The presence of queuing blurs the boundary between a processes and channels and between processes and **GMS**. This insight is crucial in reasoning about failures.

Some events have an actual or potential causal relations between them. This is captured by a *partial order* on events. The queuing event of a packet always precedes its dequeuing event. In addition, the events at a specific process are linearly ordered, capturing the assumption of a single threaded processing of trigger events. In fact, at every process the sequence of events can be broken into intervals composed of a single trigger followed by a finite sequence of the zero or more side effects that are caused by the processing of the trigger. We refer to such a sequence as a *transaction*.

The structure of the transactions that compose the event set of each process is determined by **PROTOCOL**. Each trigger event causes a **PROTOCOL**-specific callback to be executed. The callback can

generate an arbitrary number of queuing events.

While the number of events may be infinite over the life of a process, there is only a finite number of events preceding each particular event.

2.3 The Formal Model

2.3.1 Ingredients of the model

\mathbb{P}

The set of all *processes*.

\mathbb{P}_h

The set of all halting processes. $\mathbb{P}_h \subset \mathbb{P}$.

\mathfrak{V}

The number of views. $0 < \mathfrak{V} \leq \infty$

\mathbb{S}_i **where** $i < \mathfrak{V}$

The set of members of the i^{th} view. \mathbb{S}_i is a finite set of processes. We will freely refer to both \mathbb{S}_i and its index i as a "view".

\mathbb{K}

The set of all packets. A packet k has a *content* which is denoted by $\text{cont}(k)$ and is protocol and application specific. A packet is *sent* from a source process and *received* by a target process. Due to faults, a packet may fail to be sent or received. So in general a packet is either sent and received, or sent and not received, or not sent. In the second case, where the packet is sent but not received, we say that the packet is *dropped*.

By abuse of notation we will say $k = c$ when we mean $\text{cont}(k) = c$.

\mathbb{F}

The set of all **GMS** notifications. For each view i and each process P there is at most one notification for view change i that is supposed to be received at P . We denote this notification by $v_i(P)$. A notification has a *content* which is made up of the view change type (join or removal), the identity of the process that is joining or is being removed, and possibly the identity of the parent process (in case of a join). We denote the content of the notification by $\text{cont}(v_i(P))$. The possible contents for a notification are:

$\mathbf{n}_{\text{REM}}\langle \text{pid} \rangle$

Process pid is removed.

$\mathbf{n}_{\text{JOIN}}\langle \text{pid}, \text{p_pid} \rangle$

Process pid is joining, and its parent is the existing member p_pid .

$\mathbf{n}_{\text{START}}\langle \text{pid} \rangle$

You (the recipient of the notification) are the new member of the group, and your identifier is pid .

\mathbf{n}_{STOP}

You (the recipient of the notification) are no longer a member of the group.

Due to faults, a notification may fail to be received by its target process. In such a case we say that the notification is *dropped*.

By abuse of notation we will say $v_i(P) = c$ when we mean $\text{cont}(v_i(P)) = c$.

A

The set of all message broadcast requests from **APP**. An message broadcast request r has a *content* which is denoted by $\text{cont}(r)$ and is application specific. Since these requests are generated locally we assume that they are never dropped.

\overrightarrow{PQ}

The unidirectional channel from a *source* process P to a *target* process Q . A channel is a set of packets. $\overrightarrow{PQ} \subset \mathbb{K}$.

\mathbb{E}, \prec

The set of all events, partially ordered by the \prec order relation. If $e \prec f$ we say that event e *precedes* event f , or that e is *earlier* than f and f is *later* than e .

The \prec relationship is a *weak* partial order, meaning that in some cases both $e \preceq f$ and $f \preceq e$ can hold at the same time for $e \neq f$. We will indicate such cases by $e \asymp f$ and say that e and f are *contemporaneous*.

\mathbb{E}_P

The set of all events that occur at process P . If this set is empty we say that P is *uninitialized*.

\mathbb{A}_P

The set of all requests that **APP** issues at process P .

GMS

A Group Membership Service that delivers view change notifications to the processes.

PROTOCOL

A protocol that determines how triggers are processed and what side effects they create.

APP

A user application that generates message broadcast requests at various processes.

Events in the model

k^{QU}

The packet $k \in \overrightarrow{PQ}$ is appended to the send queue of the channel. This event occurs at P .

k^{PR}

The packet $k \in \overrightarrow{PQ}$ is removed from the receive queue of the channel and processed. This event occurs at Q .

$v_i(P)^{\text{PR}}$

The notification $v_i(P)$ is removed from the receive queue at process P and processed. This event occurs at P .

r^{PR}

The message broadcast request $r \in \mathbb{A}_P$ is removed from the APP receive queue and processed. This event occurs at P .

The set \mathbb{E}_P includes all the packets queued by P , all the packets dequeued by P , all the view notifications dequeued by P , and all the message broadcast requests dequeued at P :

$$\mathbb{E}_P = \left\{ k^{\text{QU}} \in \mathbb{E} \mid \exists Q(k \in \overrightarrow{PQ}) \right\} \cup \left\{ k^{\text{PR}} \in \mathbb{E} \mid \exists Q(k \in \overrightarrow{QP}) \right\} \\ \cup \{v_i(P)^{\text{PR}} \in \mathbb{E}\} \cup \{r^{\text{PR}} \in \mathbb{E} \mid r \in \mathbb{A}_P\}$$

2.3.2 The PROTOCOL interface

We mentioned in the introduction that each notification; dequeued packet; and message broadcast request has to be processed somehow. The processing is mostly controlled by PROTOCOL which consists of a small number of non-blocking calls that we will describe below.

When APP wishes to broadcast a message m it invokes the PROTOCOL call `protBroadcast(m)`.

When a brand new process group Grp is initialized, the PROTOCOL call `protStart(Grp, P)` must be invoked manually at each member process $P \in Grp$.

When a notification n is dequeued at a process P from the notification queue, some pre-processing has to occur before PROTOCOL can take over. First of all removal notifications have to be separated from join notifications.

If the notification is the removal notification of a process R then the PROTOCOL call `protRemove(R)` is invoked.

If the notification is a join notification of a process J with parent E , the process P has to determine whether it is the designated parent of the joining process. If P is not the designated parent ($P \neq E$) then it invokes the PROTOCOL call `protJoin(J, E)`. If P is the parent then it must first fork the new process and call `protRun(J)` in the child process to initialize J . This is summarized in the

pseudo-code below.

Procedure doNotification(n)

```

Input:  $n$  is the notification being processed
if  $n = n_{\text{REM}}\langle R \rangle$  then
    protRemove( $R$ );
end
else if  $n = n_{\text{JOIN}}\langle J, E \rangle$  and  $E \neq \text{self}$  then
    protJoin( $J, E$ );
end
else
    // the local process is the parent
    switch fork() do
        case parent:
            // this block is executed when fork() returns in the parent
            protJoin( $J, E$ );
        endsw
        case child:
            // this block is executed when fork() returns in the child
            protRun( $J$ );
        endsw
    endsw
end

```

When a received packet k is dequeued at a target process T from the receive queue of an incoming channel \overrightarrow{ST} , the process invokes the **PROTOCOL** call **protPacket**(k, S).

All in all the following six non-blocking calls must be implemented by **PROTOCOL**:

protBroadcast(m)

This call is issued by **APP** when it wishes to broadcast a message m .

protStart(**roster**, P)

This call is issued manually at each initial member process when the group is initialized at the start of view zero. **roster** is the set of initial members and $P \in \text{roster}$ is the identifier of the process at which the call is issued. While **GMS** does not issue join notifications to the initial members at view zero, this call appears as if it were issued in response to such a notification.

protRun(P)

This call is issued at a new process right after it is forked from its parent. P is the identifier of the new process. At the moment of forking the new process is identical to its parent and this call is the means by which the new process acquires an independent identity. While **GMS** does not issue a join notification to the joining member, this call appears as if it were issued in response to such a notification.

protRemove(P)

This call is issued in response to a removal notification from **GMS**. P is the identifier of the removed process.

protJoin(P, E)

This call is issued in response to a join notification from GMS. P is the identifier of the joining process and E is the identifier of its parent.

protPacket(k, S)

This call is issued when a packet is dequeued from a receive queue. k is the received packet and S is the process identifier of the sender of the packet.

2.3.3 The APP interface

The user application APP runs in its own thread at each process. Its only means for communicating with other processes is the protBroadcast call. With the help of PROTOCOL, APP can have view change notifications and messages from other processes delivered to it. In order to facilitate these deliveries, APP must implement a small number of non-blocking callbacks that are executed by PROTOCOL within the context of PROTOCOL calls. As a result these callback execute outside the context of the main APP-thread.

We assume that these callbacks are used by APP to manage an opaque (APP-dependent) data structure ReplicatedData. This data structure can be manipulated by the callbacks, but the main APP thread can only read that structure and not change it. In order to allow ReplicatedData to be initialized APP must provide an initialization callback.

The callbacks are not able to invoke the protBroadcast call. Only the main thread can do that.

Specifically, APP must obey the following rules:

1. APP must implement the following callback functions:
 - GroundState(): This call creates the initial value of ReplicatedData. It is called when a member of view zero is initialized, thus guaranteeing that the replicated data will start with coherent values at all the initial processes (the meaning of "coherent" here is APP-specific. It can simply mean "identical", but it can also indicate a more complex relationship.)
 - ApplyMessage(msg, originator): This call applies a message msg from process originator to ReplicatedData. One possible way to apply the message is to append it to a delivery log (see [4]).
 - ApplyJoin(pid): This call applies the notification that a process with identity pid has joined the group. In [4] such notifications are appended to the delivery log.
 - ApplyRemoval(pid): This call applies the notification that the process with identity pid has been removed from the group. In [4] such notifications are appended to the delivery log.
2. APP must implement a Main(pid) function. This is the main application thread. Its only parameter is the local process identity. This implies that when APP is started at a new process, it has no context except for the local process identity and the information that is available to it through ReplicatedData.

3. The `Main()` function of APP may invoke the `protBroadcast` procedure but the callbacks may not.
4. The `Main()` function has read-only access to `ReplicatedData`. It may manage additional data outside of `ReplicatedData` without any restrictions.
5. The callbacks listed above have read and write access to `ReplicatedData`. They have access to no other information. In particular they do not know the identity of the local process.
6. The `Main()` function runs in its own thread. The callbacks are called in the context of a critical section, and therefore must not block.

2.3.4 Model axioms and histories

View Interval Axiom

A process P is a member of at least one view, and the set of views of which it is a member is an unbroken interval, called the *view interval*, which is either finite or infinite. Formally

$$\{i | P \in \mathbb{S}_i\} = \{i | j(P) \leq i < r(P)\}$$

If $j(P) > 0$ then we say that $j(P)$ is the *join view* of P . If $j(P) = 0$ then we say that P is *original*.

If $r(P) < \mathfrak{V}$ then we say that $r(P)$ is the *removal view* of P . If $r(P) = \mathfrak{V}$ then we say that P is *not removed*.

View Change Axiom

View zero, which contains the initial group members, is finite. Every subsequent view differs from its predecessor by the addition or removal of a single process. As a result, each view other than view zero is the join view or the removal view of exactly one process.

Channel Axiom

Every packet belongs to exactly one channel. If $k \in \overrightarrow{PQ}$ we say that P is the *source* of k and Q is the *target* of k .

Packet Event Axiom

Every packet $k \in \overrightarrow{PQ}$ has a single queuing event $k^{\text{QU}} \in \mathbb{E}_P$ and at most one dequeuing event $k^{\text{PR}} \in \mathbb{E}_Q$. If the packet has a dequeuing event then

$$k^{\text{QU}} \prec k^{\text{PR}}$$

in other words k is queued (at the sending process) before it is dequeued (at the receiving process).

If $k_1 \neq k_2$ then $k_1^{\text{PR}} \neq k_2^{\text{PR}}$ and $k_1^{\text{PR}} \neq k_2^{\text{QU}}$, wherever these events exist.

If $e = k^{\text{QU}}$ then the set of packets $M_e = \{k' | k'^{\text{QU}} = e\}$ is finite. M_e is called the *multicast set* of e . All the packets in M_e have identical contents and share the same source P , but they must all have different targets. In other words, no two of them belong to the same channel. The set of target processes $T_e = \{Q \mid \exists k' \in M_e (k' \in \overrightarrow{PQ})\}$ is called the *target set* of e .

Packet Order Axiom

Channels are FIFO. Precisely, if

- packets k and k' belong to the same channel
- $k^{\text{QU}} \prec k'^{\text{QU}}$
- k'^{PR} exists

then k^{PR} exists and $k^{\text{PR}} \prec k'^{\text{PR}}$.

GMS Axiom

The following **GMS** notifications exist in the model

- For each process P and each $j(P) < i < r(P)$ there is exactly one notification $v_i(P)$.
 - If i is the join view of process J (with parent E) then $v_i(P) = \mathbf{n}_{\text{JOIN}}\langle J, E \rangle$
 - If i is the removal view of process R then $v_i(P) = \mathbf{n}_{\text{REM}}\langle R \rangle$.
- For each process P with $j(P) > 0$ there is exactly one notification $v_{j(P)}(P) = \mathbf{n}_{\text{START}}\langle P \rangle$
- For each process P with $r(P) < \mathfrak{V}$ there is exactly one notification $v_{r(P)}(P) = \mathbf{n}_{\text{STOP}}$

We also add the following artificial notifications that do not relate to actual **GMS** notifications:

- For each original process P we add a notification $v_0(P) = \mathbf{n}_{\text{START}}\langle P \rangle$
- For each process that halts and is not removed we add a notification $v_{\mathfrak{V}}(P) = \mathbf{n}_{\text{STOP}}$

Notification Event Axiom

If $v_i(P)$ exists, there is at most one $v_i(P)^{\text{PR}}$ event. If $i = 0$ and P is original then $v_i(P)^{\text{PR}}$ exists.

If $v_{j(P)}(P)^{\text{PR}}$ exists, we will use the shorthand $P_{\text{RUN}} \equiv v_{j(P)}(P)^{\text{PR}}$

If $v_{r(P)}(P)^{\text{PR}}$ exists, we will use the shorthand $P_{\text{HLT}} \equiv v_{r(P)}(P)^{\text{PR}}$

Notification Order Axiom

View notifications are dequeued in order. Precisely, if $j(P) \leq i < i' \leq r(P)$ and $v_{i'}(P)^{\text{PR}}$ exists then

1. $v_i(P)^{\text{PR}}$ exists.
2. $v_i(P)^{\text{PR}} \prec v_{i'}(P)^{\text{PR}}$

Parent Axiom

If J joins in view $j(J) > 0$ and E is its parent and J_{RUN} exists then $v_{j(J)}(E)^{\text{PR}}$ also exists and $v_{j(J)}(E)^{\text{PR}} \prec J_{\text{RUN}}$. In other words a new process would not instantiate unless its parent has processed the notification that announces its joining but we model the two events as being contemporaneous.

Process Order Axiom

The precedence order \prec is a linear order at each process P . In other words any two events in \mathbb{E}_P are \prec -comparable.

Process Liveness Axiom

A process P does not queue a packet to send to another process Q unless Q appears live to P . In other words, if $k \in \overrightarrow{PQ}$ then the following two conditions must be met:

1. $Q \in \mathbb{S}_{j(P)}$ or, $v_{j(Q)}(P)^{\text{PR}} \in \mathbb{E}_P$ and $v_{j(Q)}(P)^{\text{PR}} \prec k^{\text{QU}}$
2. If $v_{r(Q)}(P)^{\text{PR}} \in \mathbb{E}_P$ then $k^{\text{QU}} \prec v_{r(Q)}(P)^{\text{PR}}$

Take note that this axiom is a statement about the behavior of **PROTOCOL**, since its callbacks are the only elements of the model that generate queuing events.

Piggyback Axiom

Packets are processed in the same or higher view than the one at which they are queued¹. In other words, if $k \in \overrightarrow{PQ}$ and k^{PR} exists, then for any i , if $i \leq j(P)$ or $v_i(P)^{\text{PR}} \prec k^{\text{QU}}$ then $i \leq j(Q)$ or $v_i(Q)^{\text{PR}}$ exists and $v_i(Q)^{\text{PR}} \prec k^{\text{PR}}$.

Self Channel Axiom

Packets on self channels are processed early. If P is a process, $k \in \overrightarrow{PP}$ is a packet and i is a view such that $v_i(P)^{\text{PR}}$ exists and $k^{\text{QU}} \prec v_i(P)^{\text{PR}}$ then k^{PR} exists and $k^{\text{PR}} \prec v_i(P)^{\text{PR}}$.

Request Event Axiom

If request $r \in \mathbb{A}_P$ exists, there is at most one r^{PR} event.

Order Foundation Axiom

The \prec order in \mathbb{E}_P is *very well founded*, meaning that every event is preceded by only a finite number of earlier events. Formally, for any event $e \in \mathbb{E}_P$:

$$|\{f \in \mathbb{E}_P \mid f \prec e\}| < \infty$$

If $\mathbb{E}_P \neq \emptyset$ then P_{RUN} exists and is the first element of \mathbb{E}_P .

If P_{HLT} exists then is the last element of \mathbb{E}_P .

Minimal Order Axiom

The order \prec is the minimal order generated by the order relations at each process and by the orders stipulated by the Packet Event Axiom and the Parent Axiom.

¹For this axiom to hold the implementer has to "piggyback" the latest view change information on every packet that is sent, in case this information is missing at the receiving side

First Halting Axiom

A halting process P has a finite event set. In other words, if $P \in \mathbb{P}_h$ then $|\mathbb{E}_P| < \infty$.

Second Halting Axiom

Let P be a process and let i be a finite integer in the interval $j(P) \leq i \leq r(P)$. Then

- If $v_i(P)$ is dropped then $v_i(P)^{\text{PR}}$ does not exist.
- If P does not halt and $v_j(P)$ is not dropped for any $j \leq i$ then $v_i(P)^{\text{PR}}$ exists. In other words if P does not halt then it dequeues all the notifications that it is legally allowed to process.

Third Halting Axiom

Let P and Q be processes and let $k \in \overrightarrow{PQ}$ be a packet. Then

- If P does not halt then k is sent. In other words, a non-halting process eventually sends all the packets that it queues to its send queues.
- If k is not received then k^{PR} does not exist.
- If
 - P and Q do not halt
 - Every packet $k' \in \overrightarrow{PQ}$ where $k'^{\text{QU}} \preceq k^{\text{QU}}$ is received

then k^{PR} exists. In other words in the absence of a gap or stoppage, a packet in a channel is eventually dequeued and processed.

Fourth Halting Axiom

Let P be a process and let $r \in \mathbb{A}_P$ be an message broadcast request. If P does not halt then r^{PR} exists. In other words a non-halting process eventually dequeues and processes all of its message broadcast requests.

Definition 1. A particular vector of values $(\mathbb{P}, \mathbb{P}_h, \mathbb{K}, \mathbb{F}, \mathbb{A}, \mathbb{E}, \prec)$ that satisfies the model axioms is called a **history**.

2.4 Event Order and Time

The group computation model that we presented in the previous section does not include a notion of time. But it does include a notion of causality that is embodied by the partial order on events. Furthermore, any instance of group computation, embodied by the notion of a history, does unfold in physical time. How is physical time related to event order? intuitively, effects always follow causes in time. Also, within any finite interval of time only a finite number of events can occur. Beyond that we can say nothing. In other words, given any physical group computation in our model there should be a timestamp mapping

$$\text{time} : \mathbb{E}^H \rightarrow \mathbb{R}$$

Where H is the history of the computation, \mathbb{R} is physical time as represented by the real numbers, and $\text{time}(e)$ is the physical time at which the event e occurs. The mapping $\text{time}(\cdot)$ must have the following properties:

1. $e \asymp f \implies \text{time}(e) = \text{time}(f)$
2. $e \prec f \implies \text{time}(e) < \text{time}(f)$
3. $\left| \left\{ e \in \mathbb{E}^H \mid \text{time}(e) < r \right\} \right| < \infty \quad \text{for all } r \in \mathbb{R}$

Conversely, any arbitrary timestamp mapping $\text{time}(\cdot)$ that meets the above criterion should be realizable by a physical computation that yields the history H . One simply has to assume that the computation unfolds at each process at the speed that is dictated by $\text{time}(\cdot)$.

With that in mind, we want to show that every history can be realized by a physical computation that unfolds in physical time in a particularly convenient fashion. Namely we want to show that for every history there is a timestamp function that enjoys the additional property

4. $\text{time}(v_i(P)^{\text{PR}}) = i$ whenever $v_i(P)^{\text{PR}}$ exists

In other words, all the notification events for view i occur at exactly the same time. By constructing such a realization we will demonstrate that race conditions where different processes have different ideas about group membership can be ignored and the local view change notification events at the various processes can be collapsed into a single event. This is exactly what we intend to do in section 5.

For this kind of timing to be possible we must show that the partial order \prec is stratified by view. In other words we must show that every event e can be assigned a value $\text{view}(e)$ such that

- $e \preceq f$ only if $\text{view}(e) \leq \text{view}(f)$.
- $\text{view}(v_i(P)^{\text{PR}}) = i$.

It turns out that this can be done for any history H .

2.4.1 König's Lemma

A key tool in this and subsequent investigations of the event order relation in H is König's Lemma, a well known property of some partially ordered sets. We use the following formulation of the lemma:

Lemma (König's Lemma). *Let A be an infinite partially ordered set with a first element a_0 where the following properties hold for any element $a \in A$:*

- *a has a finite number of immediate successors.*
- *If $b > a$ then there is an immediate successor b_0 of a such that $b \geq b_0$.*

Then A contains an infinite ascending branch.

Proof. Call an element $a \in A$ *heavy* if it has an infinite number of successors. Then obviously a_0 is a heavy element. We will show that every heavy element has a heavy successor. Once we do that

we can choose a heavy successor a_1 to a_0 , a heavy successor a_2 to a_1 , etc. until we get an infinite ascending branch $a_0 < a_1 < a_2 < \dots$.

Let a be a heavy element. By assumption, each successor $b > a$ is mediated by an immediate successor of a . Since there are an infinite number of the former and only a finite number of the latter, there must be some immediate successor c of a that has an infinite number of its own successors. In other words, c is a heavy successor of a . \square

2.4.2 Stratifying events by view

We start our investigation by showing that the set of notification events of each view form a maximal semi-antichain in the partial order \prec , and that these semi-antichains partition the packet events by the view at which they occur, a notion that we will make precise.

For each finite view i in the view interval $0 \leq i \leq \mathfrak{V}$ define

$$G_i = \{v_i(X)^{\text{PR}} \mid v_i(X)^{\text{PR}} \text{ exists}\}$$

The sets G_i form a partition of all the notification events in H .

We now extend the partition to all the events. For all finite $0 \leq i \leq \mathfrak{V}$ Define

$$\begin{aligned} \hat{K}_i &= \{e \in \mathbb{E} \mid g \preceq e \text{ for some } g \in G_i\} \\ K_i &= \hat{K}_i \setminus \bigcup_{i < j \leq \mathfrak{V}} \hat{K}_j \end{aligned} \quad \text{for finite } 0 \leq i \leq \mathfrak{V}$$

We will now investigate the order relations between events in the different G_i sets. We will show that each G_i is a semi-antichain, meaning that there are no strict \prec -inequalities between its elements, and that elements in a high G_i never precede elements in a low G_i .

Lemma 1. *Suppose that there are processes P and Q and views i, j such that*

$$v_i(P)^{\text{PR}} \prec v_j(Q)^{\text{PR}}$$

Then $i < j$.

Proof. The Minimal Order Axiom implies that the relation $v_i(P)^{\text{PR}} \prec v_j(Q)^{\text{PR}}$ is derived from a sequence of parent/child relations $(v_{j(J)}(E)^{\text{PR}} \asymp J_{\text{RUN}})$ and queuing/dequeuing relations $(k^{\text{QU}} \prec k^{\text{PR}})$. We will prove the claim by induction on the number of intermediate steps in the shortest derivation.

If the derivation is immediate then the two events must occur at the same process, namely $P = Q$. In this case the Notification Event Axiom and the Notification Order Axiom imply that $i < j$ and we are done.

If the derivation is longer, look at the first step. This step can be of packet type or parent/child type.

If the first step is of packet type then there is a process R and a packet $k \in \overrightarrow{PR}$ such that k^{PR} exists, $k^{\text{QU}} \in \mathbb{E}_P$ and

$$v_i(P)^{\text{PR}} \prec k^{\text{QU}} \prec k^{\text{PR}} \prec v_j(Q)^{\text{PR}}$$

By the Piggyback Axiom this implies that $i \leq j(R)$ or $v_i(R)^{\text{PR}} \prec k^{\text{PR}}$. If $v_i(R)^{\text{PR}} \prec k^{\text{PR}}$ then we can create a shorter derivation leading from $v_i(R)^{\text{PR}}$ to $v_j(Q)^{\text{PR}}$ and conclude by induction that $i < j$.

If $i \leq j(R)$ then we can create a shorter derivation leading from $R_{\text{RUN}} = v_{j(R)}(R)^{\text{PR}}$ to $v_j(Q)^{\text{PR}}$ and conclude by induction that $j(R) < j$ and therefore $i < j$.

If the first step is of parent/child type then there is an initialized process J that is a child of P and

$$v_i(P)^{\text{PR}} \preceq v_{j(J)}(P)^{\text{PR}} \asymp J_{\text{RUN}} \preceq v_j(Q)^{\text{PR}}$$

and either the rightmost or leftmost inequalities is strict.

The left inequality implies, by the Notification Event Axiom and the Notification Order Axiom, that $i \leq j(J)$. If the inequality is strict then $i < j(J)$.

The right inequality implies by induction that $j(J) \leq j$. If the inequality is strict then $j(J) < j$.

Together these facts imply that $i < j$ and we are done. \square

Corollary 1. $G_i \subset K_i$

Corollary 2. Every event $e \in \mathbb{E}$ is a member of exactly one K_i .

Proof. It follows directly from the definition that e cannot belong to more than one K_i . The difficult part is showing that e belongs to some K_i . The event e occurs at some process Q and therefore $Q_{\text{RUN}} \preceq e$. Therefore $e \in \hat{K}_{j(Q)}$. If $\mathfrak{V} < \infty$ then there is a largest j such that $e \in \hat{K}_j$ and therefore $e \in K_j$ and we are done. To finish the proof we have to show that even when $\mathfrak{V} = \infty$ there is such a largest j .

By the Order Foundation Axiom, there is a largest j such that $v_j(Q)^{\text{PR}} \preceq e$. We will show that j is the maximal value we are looking for. Suppose there is a process P and a view i such that $v_i(P)^{\text{PR}} \preceq e$. If the derivation of this relation is immediate then $P = Q$ and by definition $i \leq j$.

For a non-immediate relation we proceed by induction on the length of the shortest derivation.

If the derivation starts with a packet type step then there is a process R and a packet $k \in \overrightarrow{PR}$ such that

$$v_i(P)^{\text{PR}} \prec k^{\text{QU}} \prec k^{\text{PR}} \preceq e$$

By the Piggyback Axiom the leftmost inequality implies that $i \leq j(R)$ or $v_i(R)^{\text{PR}} \prec k^{\text{PR}}$. If $v_i(R)^{\text{PR}} \prec k^{\text{PR}}$ then we can create a shorter sequence leading from $v_i(R)^{\text{PR}}$ to e and conclude by induction that $i \leq j$.

If $i \leq j(R)$ then we can create a shorter sequence leading from $R_{\text{RUN}} = v_{j(R)}(R)^{\text{PR}}$ to e and conclude by induction that $j(R) \leq j$ and therefore $i \leq j$.

If the derivation starts with a parent/child type step then there is a child J of P such that

$$v_i(P)^{\text{PR}} \preceq v_{j(J)}(P)^{\text{PR}} \asymp J_{\text{RUN}} \preceq e$$

The lefthand inequality occurs within \mathbb{E}_P and therefore we know that $i \leq j(J)$. Also, since $J_{\text{RUN}} = v_{j(J)}(J)^{\text{PR}}$ we can conclude by induction that the righthand inequality implies that $j(J) \leq j$. Therefore $i \leq j$ and we are done. \square

Definition 2. If an event $e \in \mathbb{E}$ belongs to K_i , we say that the **view** of e is i and denote it by $\text{view}(e) = i$. It follows from the proof of Corollary 2 that all the events in \mathbb{E}_P immediately following a notification event $v_i(P)^{\text{PR}}$ share the same view i .

Next we investigate the order between the K_i sets.

Lemma 2. Let e and f be events such that $e \preceq f$. Then $\text{view}(e) \leq \text{view}(f)$.

Proof. By definition there is an event $g \in G_{\text{view}(e)}$ such that $g \preceq e$. Therefore $g \preceq f$ and therefore $f \in \hat{K}_{\text{view}(e)}$. It is easy to see from the definition of K_* that this implies $\text{view}(e) \leq \text{view}(f)$. \square

Corollary 3. The partial event order \prec is very well founded in \mathbb{E} , meaning that for each event e , the set $\{f \in \mathbb{E} \mid f \prec e\}$ is finite.

Proof. Let e be an event and let i be the view of e . Then for every preceding event $f \prec e$ with view j we have $j \leq i$ by Lemma 2. Therefore if a process P joins at a view that is higher than i then \mathbb{E}_P does not contain any predecessor of e because for any $f \in \mathbb{E}_P$ we have $P_{\text{RUN}} \prec f$ and therefore $\text{view}(f) > i$. So all the events that precede e come from early joining processes, and it follows from the View Change Axiom that there is only a finite number of such processes.

Suppose that the partially ordered set of predecessors of e (including e itself) is infinite. The Minimal Order Axiom implies that each event in the set has at most two immediate predecessors (one at the process and one at the channel. A join event of a child has one strict immediate predecessor at the parent process.) The same axiom, together with the Process Order Axiom and Order Foundation Axiom imply that every predecessor is mediated by an immediate predecessor. By inverting the direction of the order König's Lemma implies that there is an infinite decreasing sequence

$$e = e_0 \succ e_1 \succ e_2 \succ \dots$$

All the predecessors of e reside on a finite number of early joining processes, and therefore there is one process X that contains an infinite number of the events in the regression chain $e_{j_1} \succ e_{j_2} \succ \dots$. But this means that e_{j_1} is an event in \mathbb{E}_X that has an infinite number of predecessors at the same process X . This contradicts the Order Foundation Axiom. \square

2.4.3 Creating an event timeline

We want to show that the properties of the event order in H are sufficient for the creation of a timeline that meets the basic timeline criteria (1)-(3) as well as the additional criterion (4).

The naïve plan is to start by assigning $\text{time}(g) = v$ for each event $g \in G_v$. This should work thanks to Lemma 1 and because the sets G_v are finite. Then it would be tempting to squeeze all the events in $K_v \setminus G_v$ into the open time interval $(v, v + 1) \in \mathbb{R}$.

This would have worked if we could guarantee the finiteness of K_v . But we cannot do that because of a number of permissible pathological situations. For example, a process can be removed and yet not halt. Such a process could generate an infinite number of events with a fixed view. Another example is a process that is not removed, but stops receiving GMS notifications at some point. As a result all the events at the process have a fixed view beyond a certain point. A reasonable group communication protocol would eliminate such pathologies (e.g. through various timeout clocks that

would force a process to halt when a pathology is suspected.) In Section 2.5.1 we explore a set of such "reasonableness" assumptions. For now we explore the more general situation that requires more finesse in constructing the timeline.

The idea is to force the events in G_v to occur contemporaneously while allowing some events in K_v to occur indefinitely late, bounding their timing by a measure of their pathology. We measure the pathology of an event e using the notion of a view bound:

Definition 3. Let $K = \bigcup_{0 \leq i < \mathfrak{V}} (K_i \setminus G_i)$. Let $e \in K$ be an event. The **view bound** of e is

$$\text{vb}(e) = \begin{cases} \min \{w \mid \exists g(e \prec g \in G_w)\} & \text{if such views exist} \\ \mathfrak{V} & \text{otherwise} \end{cases}$$

Definition 4. K^b denotes the set of events in K of bound $b < \mathfrak{V}$. $K^{\mathfrak{V}}$ denotes the unbounded events in K .

It follows from the definition of $\text{view}()$ and from Lemma 1 that $\text{vb}(e) > \text{view}(e)$.

It follows from Corollary 3 and from the fact that the sets G_i are finite that the set K^b is finite for each $b < \mathfrak{V}$. The set $K^{\mathfrak{V}}$ may be infinite.

We can now amend our timing plan by placing the events in the finite set K^b on the time interval $(b-1, b)$ and by carefully distributing the events in $K^{\mathfrak{V}}$ between different time intervals. To do this we need the following set-theoretic lemma:

Lemma 3. Let (A, \prec) be a countable, very well founded, partially ordered set. Then the elements of A can be listed in a sequence that respects the partial order \prec . In other words, the partial order \prec can be extended to a total order of order type ω , the order type of the natural numbers.

We will prove the lemma below. We use Lemma 3 to order the elements of $K^{\mathfrak{V}}$ into a sequence

$$K^{\mathfrak{V}} = \{e^1, e^2, e^3, \dots\}$$

that respects the \prec -order.

Now we can place all the events of \mathbb{E} on a timeline.

Obviously, every event in $g \in G_v$ occurs at time v . As we mentioned before, the events of K^b occur in the time interval $(b-1, b)$. Lemma 3 guarantees that the events in K^b can be arranged along that interval in a way that respects the \prec -order. In order to reserve some free time to schedule $K^{\mathfrak{V}}$ events in the interval, we restrict the events in K^b to the smaller interval $(b-1, b-1/2)$.

Now we can place the events of $K^{\mathfrak{V}}$ along the timeline. To make matters simpler, we will place all of the events in this set at times of the form $\text{time}(e^i) = n_i + 3/4$ where $n_i \in \mathbb{N}$. We do that inductively by defining:

$$\text{time}(e^i) = \max \{ \lfloor \text{time}(f) \rfloor \mid f \prec e^i \} + 7/4 \quad (\lfloor x \rfloor \text{ stands for the integer part of } x)$$

This inductive formula is well defined because each e^i has a finite number of predecessors (Corollary 3) and because every predecessor of e^i in $K^{\mathfrak{V}}$ must come earlier in the sequence than e^i and therefore has its time already defined.

Our construction of a timeline for \mathbb{E} clearly satisfies conditions (1), (3) and (4). We have to demonstrate that condition (2), which stipulates that the timeline respects the \prec -order, is also satisfied. Take any two events e_1, e_2 such that $e_1 \prec e_2$.

If $e_2 \in K^{\mathfrak{V}}$ then by construction $\text{time}(e_1) < \text{time}(e_2)$. Henceforth we will assume that if $e_2 \in K$ then $\text{vb}(e_2) < \mathfrak{V}$.

If $e_1 \in G_i$ and $e_2 \in G_j$ then Lemma 1 implies that $\text{time}(e_1) = i < j = \text{time}(e_2)$.

If $e_1 \in G_i$ and $e_2 \in K$ then Lemma 2 implies that $\text{view}(e_2) \geq i$ and therefore $\text{vb}(e_2) > i$. Since we can also assume $\text{vb}(e_2) < \mathfrak{V}$ it follows that $\text{vb}(e_2) - 1 < \text{time}(e_2) < \text{vb}(e_2)$. Therefore

$$\text{time}(e_1) = i \leq \text{vb}(e_2) - 1 < \text{time}(e_2)$$

If $e_1 \in K$ and $e_2 \in G_j$ then by definition $\text{vb}(e_1) \leq j < \mathfrak{V}$ and therefore

$$\text{time}(e_1) < \text{vb}(e_1) \leq j = \text{time}(e_2)$$

We are left with the case where $e_1, e_2 \in K$. The order $e_1 \prec e_2$ implies $\text{vb}(e_1) \leq \text{vb}(e_2)$ and we are assuming that $\text{vb}(e_2) < \mathfrak{V}$. If there is a strict inequality $\text{vb}(e_1) < \text{vb}(e_2)$ then

$$\text{time}(e_1) < \text{vb}(e_1) \leq \text{vb}(e_2) - 1 < \text{time}(e_2)$$

If there is equality then both e_1 and e_2 belong to the same set $K^{\text{vb}(e_1)} = K^{\text{vb}(e_2)}$ and their timing matches their order by construction.

Proof of Lemma 3. The set A is countable. Let the bijection $h : \mathbb{N} \rightarrow A$ be an arbitrary ordering of A into a "bad" sequence (one that does not necessarily extend the \prec -order.)

Let $c_* = c_1, c_2, c_3, \dots$ be an arbitrary sequence of natural numbers where each number appears an infinite number of times (for example one could use the sequence $1, 2, 1, 3, 2, 1, 4, 3, 2, 1, \dots$)

To create the sequential extension of the \prec -order, Start with an empty "good" sequence. We will append the elements of A to the good sequence one at a time using the following infinite process. At the n^{th} step, look at the c_n^{th} element of the bad sequence, namely the element $h(c_n)$. Now do the following:

1. If $h(c_n)$ has already been appended to the good sequence, do nothing.
2. If some predecessor $a \prec h(c_n)$ has not yet been appended to the good sequence, do nothing.
3. If $h(c_n)$ has not yet been appended to the good sequence, but all of its predecessors in the \prec -order had been appended, append $h(c_n)$ to the good sequence now.

The good sequence that this procedure generates has some obvious good properties. It includes at most one copy of each element of A , and the sequence order respects the \prec -order. We just have to show that every element of A is eventually appended to the good sequence.

Assume instead that some element of A is never appended to the good sequence. Look at the subset $A_0 \subset A$ of elements that are never appended. This subset is not empty by assumption. Since the \prec -order on A is very well founded, there must be a \prec -minimal element in A_0 . Suppose that this minimal element is $a_0 = h(j)$. Since the \prec -order is very well founded, the element a_0 has a finite number of \prec -predecessors a'_1, a'_2, \dots, a'_k .

By assumption all of these predecessors are appended to the good sequence, and this must happen by some finite step n_0 . By our assumption on the sequence c_* there is some high number $n > n_0$

such that $c_n = j$. At step n the element $a_0 = h(j) = h(c_n)$ will be inspected and it will be found that all of its predecessors have already been appended to the good sequence. As a result a_0 will be appended at this point, contrary to our assumption. \square

2.5 Understanding Faults

A history represents a possible computation by a group of processes in our model. Such a computation can be plagued by different types of stop faults - halting processes, dropped packets and dropped GMS notifications. Our goal is to demonstrate that by adding a small number of reasonable restrictions to the model, we will be able to analyze all histories in our model under the assumption that all faults are eliminated except for one simple fault - the simultaneous halt of all the processes in the group. This is a very desirable property, because it means that the whole group behaves the same way a single process would - albeit at a much higher performance and availability level.

The fault simplifications techniques that we propose all involve the use of timers, timeouts and voluntary process halts when timeouts occur. But we are going to keep our computational model asynchronous and avoid introducing the notion of time explicitly. We achieve this by introducing axiomatic correlations between certain faults. These correlations can be forced to occur through the use of timers and timeouts, but these implementation details are not part of the model itself.

Let us start by looking at notifications. Suppose that P fails to dequeue a notification that P is entitled to. How can that happen? According to the Second Halting Axiom either some $v_j(P)$ is dropped or P halts. So the blame can be assigned to P or to GMS. If we want to simplify the fault model and eliminate GMS faults, we must shift the blame to P . But this is not possible unless P halts. A reasonable implementation will not allow a process P to continue running indefinitely when GMS becomes unresponsive. At some point P will timeout and halt voluntarily. If that happens then we have a hope of shifting the blame to P and away from GMS.

A similar observation can be made about packets. Suppose there is a packet $k \in \overrightarrow{PQ}$ that is queued by P but not dequeued by Q . How does that happen? According to the Third Halting Axiom either P halts (which opens the possibility that k is never sent), or Q halts (which opens the possibility that k is never dequeued even if it is received) or else k - or some packet ahead of k - is dropped. So the blame can be assigned to P , to Q or to the channel \overrightarrow{PQ} . If we want to simplify the fault model and eliminate \overrightarrow{PQ} faults, we must shift the blame either to P or to Q . But this is not possible unless one of them halts. A reasonable implementation will not allow P and Q to continue running indefinitely when the channel between them becomes unresponsive. At some point one of them must be successfully evicted, and failing that, both of them will timeout at some point and halt voluntarily. If that happens then we have some hope of shifting the blame to P or Q and away from the channel between them.

This built-in ambiguity of the boundary between a process and its channels and membership service allows us to reinterpret the root cause of visible faults. If a packet is not dequeued, we can shift the blame from the channel to the process and vice versa. If a notification is not dequeued we can shift the blame from the membership service to the process and vice versa. But there are limits to this blame shifting game. If a process halts, we cannot keep on blaming it forever. If other processes keep sending it packets indefinitely, then at some point the channels leading to the halted process must start dropping packets since there is no process on the target side to receive them. If

the membership service keeps attempting to inform the halted process of view changes, and never succeeds in evicting it, then its notifications will have to start dropping at some point.

Well implemented processes will not, however, keep attempting to send packets to a process that has already halted a long time ago. A well implemented membership service will not keep a halted process as a member in view after view indefinitely. A well implemented process will monitor its channels and GMS and attempt to detect problems and remove them, and at some point it will have to give up and halt.

If everyone behaves well enough then perhaps the blame for all the faults in the system can be shifted to the processes.

The process faults themselves cannot be eliminated entirely, but they can be greatly simplified. When a process halts, there can be an arbitrary, finite number of unsent packets in its send queues, and an arbitrary finite number of packets, notifications and message broadcast requests in its receive queues. The halt can occur in the middle of the processing of a trigger event, when some of the side effect events have already occurred while others have not. This is very different than what one would expect to happen when a process is removed in an orderly fashion. In an orderly removal you would expect a notification to go to all the processes, including the removed process itself, and you would expect the group to enter a quiescent period during which all on going tasks are completed and all packets in flight are received and processed while the application execution is put on hold so it does not create any new work. Only once the group is fully quiescent will the removed process halt, the group reconfigure itself, and finally resume its normal work under the new configuration.

We will demonstrate that under reasonable assumptions, a general process halt can be simplified to look like an orderly removal - with the aforementioned exception of a simultaneous halt of the whole group. This means that with this one exception, we can view any process halt as the *consequence* of a planned process removal, rather than as a failure that was the *cause* of a removal. In other words, we can make process failures go away entirely, with the one notable exception of a simultaneous system-wide failure.

First of all we can assume that a halting process completes the execution of its final transaction before halting (see Section 2.2 for a definition of transactions). This can affect its send queues and its internal state, but it does not affect any or the remaining processes as long as the packets that the transaction generates remain stuck in a send queue. Also as long as the newly generated packets are not sent out they do not adversely effect the perceived reliability of the outbound channels of the process.

A further simplification can be achieved by emptying out the receive queues of the process. This is more tricky. The basic idea is pretty simple - just assume that the process dequeues and processes all the items in its receive queues before it halts. This can generate a lot of side effects, but as long as the side effects are stuck in their respective send queues, the surviving members of the group will not be affected.

A final simplification can be achieved by emptying out some of the send queues. This is the most tricky part since it allows a process that is presumably already halted to affect other processes by sending them additional packets. In a sense the halted process can alter the information held by other processes and change history "from the grave". This can be made acceptable if we limit ourselves to only sending packets bound to processes that appear to have halted even earlier. This should confine any new information to the world of the dead and prevent it from altering history.

There are three additional complications.

The first and easiest complication is that the order of processing must comply with the Piggyback Axiom.

The second and more pernicious complication is that the processing must comply with the Self Channel Axiom. The issue is that while most side effects can be left stuck in a send queue or sent "downstream" to an already halted process, the latter axiom sometimes forces packets on the self channel to be sent, received and processed, thus generating more side effects, which themselves could result in more packets on the self channel, ad infinitum. We cannot allow that to happen because this violates the First Halting Axiom. This problem does not go away by itself. But it does not arise if **PROTOCOL** is implemented in a reasonable fashion and does not generate infinite loops for itself in the absence of external stimuli. If the protocol meets this *vacuum convergence* condition, then we can assume that a halting process leaves no unprocessed or partially processed items.

The third and perhaps subtlest complication is that processes can give birth to child processes before they halt, so even if a packet is bound to a target that halts earlier than its source did, the information that is conveyed by that packet may live on inside a child of the target process - and the child could live indefinitely. This complication is bound together with a more general issue of race conditions that can arise when a process births a child. As with the previous complication, these race condition issues cannot be wished away. Instead we must require that **PROTOCOL** behave in a way that prevents race conditions from occurring. An additional but less critical simplifying requirement is that processes that fail to initialize must not be chosen by **GMS** to be the parents of child processes.

These observations lead us to consider a sub-class of histories that arise from *conforming models*, which are models that assume well behaving processes; a well behaving membership service; and a well behaving **PROTOCOL** layer. We will see that for such histories it is possible to drastically simplify the faults that need to be considered.

2.5.1 Conforming models and conforming histories

We claimed that under some assumptions of "reasonableness" of the way the drivers and **PROTOCOL** are implemented in a model we have a hope of simplifying the behavior of faults. We are going to make this notion exact by introducing a number of new axioms that are less general than the axioms of 2.3.4 but represent behaviors that would be expected from a reasonable implementation. As we mentioned before, implementing these reasonable behaviors requires timers, but the new axioms preserve asynchrony by describing the model simplifications as mere correlations between faults

We start with a few definitions.

Definition 5. A **stunted history** is a history where the number of processes is finite and all of them halt.

Definition 6. A **finite channel** is a channel where only a finite number of packets are dequeued. In other words a channel \overrightarrow{PQ} is finite if $\left| \left\{ k \in \overrightarrow{PQ} \mid k^{\text{PR}} \in \mathbb{E} \right\} \right| < \infty$.

Conforming Channel Axiom

If \overrightarrow{PQ} is finite, then either P is removed or Q halts.

Conforming Packet Axiom

A process P does not dequeue packets from process Q after a removal notification for Q is dequeued by P .

Conforming Notification Axiom

If any notification $v_i(P)$ is dropped then P halts.

Conforming GMS Axiom

If a process halts then it has a finite view interval.

Conforming Parent Axiom

If a process is uninitialized then it has no child processes.

Conforming Halt Axiom

If a process is removed then it halts.

Definition 7. A **conforming history** is a history that satisfies all the conforming axioms.

2.5.2 Fault equivalence**Definition 8.**

A **downstream channel** is a channel \overrightarrow{PQ} where $r(P) > r(Q)$ or $P = Q$. In other words it is either a self channel or else it is a channel whose target is removed from the group before its source is removed. A **upstream channel** is a channel \overleftarrow{PQ} where $r(P) \leq r(Q)$ and $P \neq Q$. A **upstream packet** or **downstream packet** is a packet that belongs to an upstream or downstream channel, respectively.

Definition 9.

A **vacuum event** is an event e at a halting process that has no lasting effect on the history. To be precise, e is a vacuum event if its successor events

$$\{f \mid f \succeq e\}$$

form a finite set, and all them occur at halting processes.

A **vacuum packet** is a packet k with a vacuum queuing event k^{qu} .

Obviously, all the successors of a vacuum event are vacuum events as well.

Definition 10. Two histories are said to be **fault equivalent** if they are identical with the following exceptions:

- packets that are sent in one need not be sent in the other. Packets that are received in one need not be received in the other.

- notifications that are dropped in one need not be dropped in the other.
- the two histories have the same non-vacuum events.
- the two histories have the same non-vacuum packets.

Definition 11. A history H is called **lossless** if

- H has no dropped packets
- H has no dropped notifications
- All upstream channels clear their receive queues. In other words all received upstream packets are dequeued.
- All downstream channels clear their send queues. In other words all queued downstream packets are sent.

2.5.3 The Vacuum Loop and vacuum closure

Definition 12. Let H be a lossless history and let P be any halting process in H . If $j(P) > 0$ let E be the parent of P and assume that $v_{j(P)}(E)^{\text{PR}}$ exists. Look at P at the moment that it halts. Let v be the highest view processed by P , namely the value of the highest notification $v_v(P)$ that was processed by P before it halted. By the Order Foundation Axiom such a maximal view always exists unless $\mathbb{E}_P = \emptyset$. In the latter case set $v = j(P) - 1$.

Look at the following loop, called the **vacuum loop**:

1. If the event P_{HLT} exists at the end of \mathbb{E}_P
 - remove it
 - return the notification $v_{r(P)}(P)$ to the **GMS** receive queue
 - set $v = r(P) - 1$
2. Finish processing the current item. If any new packet multicasts are generated, append the respective packet queuing events at the end of \mathbb{E}_P . Queue any new upstream packets to their respective send queues and declare them to be unsent and unreceived. Queue any new downstream packets to their respective receive queues and declare them to be sent and received.
3. If $v \geq j(P)$ and there are any requests in the **APP** receive queue, dequeue and process them one by one. For each request, append the generated request dequeuing event at the end of \mathbb{E}_P . If the processing of a request creates any side effects, append the resulting packet queuing events at the end of \mathbb{E}_P . Queue any new upstream packets to their respective send queues and declare them to be unsent and unreceived. Queue any new downstream packets to their respective receive queues and declare them to be sent and received.
4. If $v = r(P) - 1$, dequeue and process all the packets in all the receive queues of channels \overrightarrow{QP} where $Q \neq P$. For each packet, append the corresponding dequeuing event to \mathbb{E}_P ; and process the packet. If the processing of a packet creates any side effects, append the resulting packet queuing events at the end of \mathbb{E}_P . Queue any new upstream packets to their respective send

queues and declare them to be unsent and unreceived. Queue any new downstream packets to their respective receive queues and declare them to be sent and received.

5. If the receive queue of the self channel $\overrightarrow{P\dot{P}}$ is empty, proceed to step (6). Otherwise, dequeue all the packets there. For each packet, append the corresponding dequeuing event to \mathbb{E}_P ; and process the packet. If the processing creates any side effects, append the resulting packet queuing events at the end of \mathbb{E}_P . Queue any new upstream packets to their respective send queues and declare them to be unsent and unreceived. Queue any new downstream packets to their respective receive queues and declare them to be sent and received. Repeat step (5) until the receive queue of the self channel is empty.

6. Increment v

- if $v = r(P)$, append a P_{HLT} event to \mathbb{E}_P and exit.
- dequeue the $v_v(P)$ notification and process it. Append the $v_v(P)^{\text{PR}}$ event to \mathbb{E}_P . If $v = j(P) > 0$, add the order relation $v_v(E)^{\text{PR}} \prec v_v(P)^{\text{PR}}$. Notice that in this case $v_v(P)^{\text{PR}} = P_{\text{RUN}}$.
- If the processing of the notification creates any side effects, append the resulting packet queuing events at the end of \mathbb{E}_P . Queue any new upstream packets to their respective send queues and declare them to be unsent and unreceived. Queue any new downstream packets to their respective receive queues and declare them to be sent and received.
- go back to step (3).

Lemma 4. Let H be a lossless history and let P be a halting process in H . Assume that P is either a member of view zero, or else the parent of P processes the notification of the joining of P ($v_{j(P)}(E)^{\text{PR}}$ exists, where E is the parent of P).

Suppose that the vacuum loop is run against the halting state of P for a finite number of steps. Then the resulting structure is a lossless history that is fault equivalent to H . Moreover, if the original history H is conforming then the extended history is conforming as well.

Proof. We start by showing that the extended structure is a lossless history. Then we show that it is fault equivalent to the original history H .

The proof that the extension of H satisfies the history axioms is by induction. Suppose that all the history axioms remain true, and the history remains lossless after going through a certain number of steps. We will show that the same remains true after running through one more step. For most axioms we do not actually need the inductive hypothesis - they are either trivially true or can be demonstrated directly. But there are a few exceptions.

The Packet Event Axiom remains true because every new packet that we create comes with a queuing event per multicast. We only add a dequeuing event to packets that are already queued (steps (4) and (5)).

The Packet Order Axiom is violated if there are packets $k, k' \in \overrightarrow{QY}$ such that $k^{\text{QU}} \prec k'^{\text{QU}}$ and k'^{PR} exists and yet k^{PR} does not exist or does not precede k'^{PR} .

Suppose that k'^{PR} already existed prior to the current step. By induction the extension of H was a history at the conclusion of the last step and therefore k'^{QU} already existed and as a result the

preceding k^{QU} event had existed as well. Therefore by induction k^{PR} had also existed and preceded k'^{PR} and we are done.

Assume therefore that k'^{PR} is generated in the current step. This implies $Y = P$. Moreover, the only steps that create packet dequeuing events are steps (4) and (5).

If k'^{PR} is generated in the current step then the packet k' is already received after the previous step. The induction hypothesis implies that k' is a downstream packet in this case, or else it would have been dequeued already due to losslessness (if we are at step (5) then k' is downstream by definition). Therefore, again by induction, losslessness implies that the packet k must have been sent (and therefore received) before the current step. Since k was queued before k' , it must have been dequeued before k' is dequeued. Therefore k^{PR} exists and precedes k'^{PR} .

The Notification Order Axiom remains true because step (6) dequeues notifications in order and without gaps.

The Parent Axiom remains true thanks to the conditions we imposed on P .

The Process Liveness Axiom is a statement about the behavior of `PROTOCOL` callbacks, which are the only part of the model that generates queuing events. Since the vacuum loop processes triggers using the appropriate `PROTOCOL` callbacks, the axiom remains valid.

Suppose that the Piggyback Axiom is violated. Then there is a packet $k \in \overrightarrow{QP}$ that we dequeue in step (4) or (5) of the vacuum loop even though there is an i such that $i < j(Q)$ or $v_i(Q)^{\text{PR}} \prec k^{\text{QU}}$ and yet $i > j(P)$ and either $v_i(P)^{\text{PR}}$ does not exist or it exists and $k^{\text{PR}} \prec v_i(P)^{\text{PR}}$.

In the case of the self-channel (step (5)) this can be easily seen to be absurd by substituting $Q = P$.

In the case of step (4) we have by definition $k^{\text{PR}} \succ v_v(P)^{\text{PR}}$ and so it must be that $i > v$. Since $v = r(P) - 1$ in this case, we have $i \geq r(P)$. So we either have $r(P) < j(Q)$ or we have $v_{r(P)}(Q)^{\text{PR}} \preceq v_i(Q)^{\text{PR}} \prec k^{\text{QU}}$. Either of these cases violate the Process Liveness Axiom which holds by induction prior to the current step.

To see that the Self Channel Axiom remains true notice that the vacuum loop dequeues a notification (in step (6)) only after it verifies that all the packets on the self channel are processed (in step (5)).

The Order Foundation Axiom is mostly trivial except for the P_{RUN} and P_{HLT} part.

If P_{RUN} does not already exist then $v = j(P) - 1$ and there was no current item to finish processing when the vacuum loop started. By induction, $\mathbb{E}_P = \emptyset$. All the steps except step (6) become no-ops and no new events are added to \mathbb{E}_P . To see that step (5) is a no-op, notice that a packet can reside in the receive queue of the self-channel only if it was previously queued to the send queue. This would show up as a queuing event which is not possible in our case.

Step (6) increments v to equal $j(P)$ and adds a $v_v(P)^{\text{PR}} = v_{j(P)}(P)^{\text{PR}} = P_{\text{RUN}}$ event as the first element of \mathbb{E}_P . This proves this case.

Step (1) of the vacuum loop removes P_{HLT} from \mathbb{E}_P and returns it to the `GMS` receive queue, if necessary. Step (6) can create or restore the P_{HLT} event. If that happens then the loop exits, leaving P_{HLT} as the last event in \mathbb{E}_P .

All the other axioms are trivial to verify. For the Second Halting Axiom one just needs to remember that we are dealing here exclusively with lossless histories, so there are no dropped notifications.

We still have to show that executing each step of the vacuum loop preserves losslessness (see Definition 11). Since we start with a lossless history we have no dropped notifications. Every packet that is created by the vacuum loop is either unsent and unreceived (if it is an upstream packet) or sent and received (if it is a downstream packet), so the loop never creates a dropped packet and guarantees the losslessness conditions with regard to upstream and downstream channels

We have to show that the extended history is fault equivalent to the original history H . This is more or less trivial by construction. All we did was add a finite number of new packets and events. All of the new events occur at the "end of history" in the sense that they do not precede any pre-existing events. All of the new events are added at P which is a halting process. Therefore by definition we only added vacuum events and vacuum packets. Any pre-existing event acquires at most a finite number of new successors and all of them occur at a halting process. Therefore all pre-existing vacuum events remain so in the extended history.

Finally, assume that H is conforming. We must check that the extended history is still conforming.

The vacuum loop does not change the set of removed processes. It does not change the set of halting processes. It does not change the view interval of any process. It adds only a finite number of dequeuing events to \mathbb{E}_P , and it does not add any dequeuing events to any other process. Therefore the extended history satisfies the Conforming Channel Axiom, the Conforming GMS Axiom and the Conforming Halt Axiom. The Conforming Notification Axiom is vacuously true because a lossless history has no dropped notifications.

The Conforming Parent Axiom holds because any uninitialized process in the extended history is uninitialized in the original history and the vacuum loop does not create any new parent/child relationships.

The Conforming Packet Axiom holds for every process other than P because H is conforming. Suppose that P processes a removal notification of a process $X \neq P$. This implies that $r(X) < r(P)$ and therefore the channel \overrightarrow{XP} is upstream. By losslessness this implies that the receive queue of the channel is empty throughout the execution of the vacuum loop, and therefore the loop does not add any new dequeuing events for packets on that channel.

If the processing of the removal notification of X occurs during the vacuum loop, then we have already shown that all the packets from X are already processed at this point. Since the vacuum loop does not add any new packets to the X -receive queue, the axiom holds in this case as well.

If $X = P$ the axiom holds because P_{HLT} is the last event in \mathbb{E}_P , according to the Order Foundation Axiom. \square

Definition 13. *A history extension of the type that is described in Lemma 4 is called a vacuum continuation of H at P . If the vacuum loop at P terminates then there is a maximal vacuum continuation of H at P , called the vacuum closure of H at P . If the vacuum closure of H at P is equal to H (meaning that the vacuum loop did not generate any new events or packets) then we say that H is vacuum closed at P . It is trivial to check that the vacuum closure of H at P is vacuum closed at P .*

Corollary 4. *Let H be a lossless history and let P be a halting process in H . If H is vacuum closed at P , then P processes all the packets that it receives and all the notifications that it is entitled to, from P_{RUN} to P_{HLT} .*

Proof. It is easy to see that the vacuum loop does not terminate as long as there is an unprocessed GMS notification that P is entitled to. This means that if P is vacuum closed then it must have processed P_{HLT} .

In order to reach the processing of P_{HLT} , the vacuum loop must pass through steps (4) and (5) while $v = r(P) - 1$. This clears all the receive queues as required. \square

2.5.4 Transactional histories and the Fault Theorem

Definition 14. A **transactional history** is a conforming, lossless history that meets the following additional restrictions:

1. All notifications are processed to completion.
2. All message broadcast requests are processed to completion.
3. All received packets are processed to completion.

Definition 15. A protocol *PROTOCOL* is **vacuum convergent** if for any conforming model in which it participates and for any halting process in any lossless history of that model the vacuum loop terminates.

We are now ready to state the principal finding of this part of the paper - the Fault Theorem.

Theorem 1 (Fault Theorem). *Let H be a conforming history, and assume that *PROTOCOL* is vacuum convergent. Then H can be extended to a fault equivalent transactional history $\text{tr}(H)$. $\text{tr}(H)$ is called the **transactional closure** of H .*

We start by showing that drops can be eliminated, and then we move on to simplifying process faults.

Lemma 5. *If a packet $k \in \overrightarrow{PQ}$ is dropped in a conforming history, then either P halts or Q halts.*

Proof. By the Third Halting Axiom, k^{PR} does not exist. By the Packet Order Axiom, for any $k' \in \overrightarrow{PQ}$ with $k'^{\text{QU}} \succ k^{\text{QU}}$, k'^{PR} does not exist either. Therefore, considering that \mathbb{E}_P and \mathbb{E}_Q are linearly ordered, the only $k' \in \overrightarrow{PQ}$ packets for which k'^{PR} exists are the ones for which $k'^{\text{QU}} \prec k^{\text{QU}}$, and since (by the Order Foundation Axiom) the \prec relation is very well founded at P , there are only a finite number of such packets. Therefore the channel is finite. By the Conforming Channel Axiom, either P is removed or Q halts. In the former case, the Conforming Halt Axiom guarantees that P halts, and we are done. \square

Lemma 6. *In a non-stunted conforming history, a process halts if and only if it is removed.*

Proof. By the Conforming Halt Axiom we know that every removed process halts. To show the converse, suppose a process P halts but is not removed. By the Conforming GMS Axiom process P must have a finite view interval. Since P is not removed, $r(P) = \mathfrak{V}$ and so \mathfrak{V} is finite. This implies that the number of processes is finite. Let Q be any process. Since P halts, the First Halting Axiom guarantees that the channel \overrightarrow{PQ} is finite. By the Conforming Channel Axiom, P is removed or Q halts. Since P is not removed, Q must halt. We conclude that there is a finite number of processes and all of them halt. In other words the history is stunted. \square

Lemma 7. *In a conforming history, a halting process P only queues a finite number of packets and only misses a finite number of packets and notifications:*

$$\begin{aligned} \left| \bigcup_{Q \in \mathbb{P}} \overrightarrow{PQ} \right| &< \infty \\ \left| \bigcup_{Q \in \mathbb{P}} \left\{ k \in \overrightarrow{QP} \mid k^{\text{pr}} \notin \mathbb{E}_P \right\} \right| &< \infty \\ |\{i \mid j(P) \leq i \leq r(P) \text{ and } v_i(P)^{\text{pr}} \notin \mathbb{E}_P\}| &< \infty \end{aligned}$$

Proof. Let P be a halting process in a conforming history. By the First Halting Axiom, P has a finite event set. Therefore only a finite number of packets are queued by P . This proves the first claim.

By the Conforming GMS Axiom, a halting process in a conforming history has a finite view interval. Therefore P is eligible only for a finite number of view notifications. This proves the third claim.

The main difficulty is with the second claim.

Let Q be a process. First we want to show that P misses a finite number of packets from Q .

If Q halts, then Q has a finite event set, therefore Q queues a finite number of packets to \overrightarrow{QP} , therefore P misses finite number of packets from Q and we are done. So assume that Q does not halt. This implies, incidentally, that the history is not stunted.

Since the history is conforming and not stunted, Lemma 6 implies that P is removed.

If $r(P) < j(Q)$ then the Process Liveness Axiom guarantees that \overrightarrow{QP} is empty and we are done. So we can assume that

$$j(Q) < r(P) < \mathfrak{V} = r(Q)$$

And therefore by the GMS Axiom the notification $v_{r(P)}(Q)$ exists. Since the history is conforming and Q does not halt, there are no dropped notifications at Q (due to the Conforming Notification Axiom) and therefore Q must process the removal notification of P .

By the Process Liveness Axiom, for every $k \in \overrightarrow{QP}$ we have $k^{\text{qu}} \prec v_{r(P)}(Q)^{\text{pr}}$. By the Order Foundation Axiom, there is only a finite number of events in \mathbb{E}_Q that precede $v_{r(P)}(Q)^{\text{pr}}$ and therefore \overrightarrow{QP} is a finite channel. Therefore, P can only miss a finite number of packets from Q .

We have established that P misses a finite number of packets from each source process. As long as only a finite number of processes queue packets targeted at P , we are done. Since P has a finite view interval, there is only a finite number of processes Q with $j(Q) < r(P)$. For any Q with $j(Q) > r(P)$ we have already established that the channel \overrightarrow{QP} is empty. \square

Theorem 2 (Lossless History Theorem). *Every conforming history is fault equivalent to a lossless conforming history that contains the same packets and events.*

Proof. Let H be a conforming history. We are going to change H into a fault equivalent lossless history by changing the faulting characteristics of notifications and packets, without adding or

subtracting any vacuum events and packets. The conforming axioms (see 2.5.1) are not affected by such changes except for the Conforming Notification Axiom. However a lossless history has no dropped notifications, so this and all other conforming axioms are going to remain valid.

We start with notifications by simply declaring that none of the notifications are dropped. There are two catches. First, we may have just added an infinite number of notifications into the notification queues of some processes. But it follows from the Conforming Notification Axiom that dropped notifications only exist at halting processes, and it follows from Lemma 7 that halting processes only have a finite number of dropped notifications, so this problem does not occur. The other catch is that we have to prove that H is still a history. There are several assertions in the Second Halting Axiom that are related to dropped notifications which we now have to verify. Suppose that some notification $v_i(P)$ is dropped in H .

- The Second Halting Axiom claims that if $v_i(P)$ is a dropped notification then $v_i(P)^{\text{PR}}$ does not exist. This assertion is not violated when we declare that $v_i(P)$ is not dropped, so we are done in this case.
- The same axiom claims that if P does not halt and $v_i(P)$ is not dropped then $v_i(P)^{\text{PR}}$ does exist. However history H is conforming, and so by the Conforming Notification Axiom P must halt. Therefore this assertion is not violated either.

We now move to packets. We make the following changes in the faulting characteristics of packets in the channel \overrightarrow{PQ} :

- We declare all the unprocessed upstream packets to be unsent and unreceived.
- We declare all the unprocessed downstream packets to be sent and received.

The same two catches apply here as well. By preventing packets from being sent we may saddle some processes with an infinite number of unsent packets. By forcing packets to be received without being processed we may be saddling some processes with an infinite number of received packets that linger in the process' receive queues indefinitely. In addition, we have to verify all the relevant axioms.

To see that we do not create an infinite number of packets that remain stuck in the send or receive queues of a process P , notice that if P halts then Lemma 7 guarantees that only a finite number of unprocessed packets exist in P 's incoming and outgoing channels and so we cannot create infinities at P . If P does not halt then the situation is a little bit more complex and we have to look at the send queues and receive queues of P separately.

It follows from the Conforming Halt Axiom that P is not removed and therefore $r(P) = \mathfrak{V}$. As a result most of the outgoing channels of P are downstream. Since we force all the unprocessed downstream packets to be sent we do not create any unsent packets on these outgoing channels. The exceptions are the channels \overrightarrow{PQ} that lead to some other process Q that is not removed. Lemma 6 implies that Q does not halt. The Conforming Channel Axiom implies that \overrightarrow{PQ} is not finite and therefore the Packet Order Axiom implies that all the packets on the channel are processed and as a result we do not change the faulting characteristics of any of packets on these channels.

On the other hand the incoming channels of P are all upstream, with the exception of the self channel \overrightarrow{PP} . Since we force all the unprocessed upstream packets to be unsent, we do not create any received-and-unprocessed packets on these channels. We have already seen that when both

ends of a channel do not halt, all the packets on the channel are processed. Therefore P processes all of the packets on its self channel and as a result we do not change the faulting characteristics of any self channel packets.

As far as axioms go, the only axiom that is related to the fault properties of packets is the Third Halting Axiom. The first part of this axiom claims that a non-halting process sends all of its queued packets. This part is not violated by our changes because we only declare a packet to be unsent if it emanates from a removed process P . By the Conforming Halt Axiom the process P halts.

The second part of the axiom claims that a packet is not processed unless it is received. We only declare a packet to be unreceived if it is not processed. Therefore this part of the axiom is not violated.

The third and last part of the axiom claims that packets keep getting processed as long as there is no impediment such as a halting source or target, or a previous unreceived packet. Suppose P and Q do not halt. The the Third Halting Axiom implies that all the packets in \overrightarrow{PQ} are sent. Lemma 5 implies that all the packets are received and as a result the Third Halting Axiom implies that all the packets in the channel are processed. Therefore we do not touch any of these packets and the third part of the axiom remains valid.

We have to show that the revised history is lossless (see Definition 11). This follows directly from our construction. We obviously do not have any dropped packets or notifications anymore. As for channels, since we declared all the unprocessed packets in upstream channels to be unsent and unreceived we have cleared all the receive queues of these channels. Similarly, since we declared all the unprocessed packets in downstream channels to be sent and received, and since all the processed packets must have been sent to begin with, we have cleared all the send queues of these channels, as required.

Our last task is to show that the new history is fault equivalent to the original history. But this is trivial since we did not add or subtract any events. \square

Proof of the Fault Theorem. Theorem 2 established that the conforming history H is fault equivalent to a lossless conforming history H_1 . In a two step process, we will improve H_1 to a fault equivalent transactional history H_3 . The intermediate histories will have the following properties:

- H_1 is conforming and lossless.
- H_2 is conforming and lossless. In addition, all processes in H_2 are initialized and process all of their notifications.
- H_3 is transactional.

The intermediate history H_2 is constructed by running the vacuum loop to completion at each halting process. As one might expect, there are some complications.

The first complication is that for Lemma 4 to apply at a process P , we must make sure that its donor E , if it exists, had dequeued the join notification of P . This can be guaranteed by traversing the processes of H by increasing join view. Since $j(P) > j(E)$, this order guarantees that we run the vacuum loop at E before we run it at P . The vacuum convergence property of **PROTOCOL** together with Corollary 4 guarantee that E will process the join notification of P before we run the vacuum loop at P .

The second complication is that we may be running the vacuum loop an infinite number of times. This only happens if H_1 is not stunted, in which case Lemma 6 implies that every halting process is removed. At each finite step Lemma 4 guarantees that the resulting structure is a conforming, lossless history that is fault equivalent to the original history H . But we have to show that all of these properties are preserved at the limit. This is mostly but not entirely trivial.

The limiting structure H_2 is a conforming history because almost all the history axioms and conforming history axioms either deal with views and view intervals, or with the events at a single process, or with events at a pair of processes, or with packets in a single channel. Any such axiom is either not affected by the vacuum loop at all, or is affected only by a finite number of applications of the vacuum loop in our infinite sequence. Therefore all of these axioms follow immediately from Lemma 4. The only exception is the Minimal Order Axiom, but this axiom is "continuous" in the sense that it naturally commutes with limits. This is because each order relationship that exists at the limit already exists after a finite number of applications of the vacuum loop.

H_2 is lossless for the same reason: the requirement that there be no dropped packets or notifications is continuous - it is fulfilled at the limit if it is fulfilled at each step. The requirements on upstream and downstream channels affect one channel at a time and each channel is affected only by the execution of the vacuum loop at the source and target of the channel.

The non-trivial part is in showing that H_2 is fault equivalent to H_1 . The argument in Lemma 4 was that the vacuum loop does not create any new non-vacuum events because it only introduces a finite number of events, all of which are vacuum events themselves. Obviously a finiteness argument of this sort cannot simply be carried over to the limit. Instead the fault equivalence of H_1 and H_2 arises from deeper roots.

Let e be any event in H_2 that has an infinite number of successor events in H_2 . The fault equivalence claim will follow if we can show that e already has an infinite number of successors in H_1 . From König's Lemma it follows that there is an infinite increasing sequence

$$B = b_1 \preceq b_2 \preceq b_3 \preceq \dots$$

of successors of e in H_2 , and we can assume that B contains no gaps, meaning that any two consecutive pair of events in $b_i \preceq b_{i+1}$ in B is a primitive relation, meaning that either

- b_i and b_{i+1} are adjacent events in a process P .
- There is a packet k such that $b_i = k^{\text{QU}}$ and $b_{i+1} = k^{\text{PR}}$.
- There is a parent/child pair of processes E/J such that $b_i = v_{j(J)}(E)^{\text{PR}}$ and $b_{i+1} = J_{\text{RUN}}$

All we need to show is that B cannot be made up exclusively of events that are outside of H_1 , namely events that are added by the extension process. We demonstrate that through a sequence of claims. We assume that B is made up by events that are added by the vacuum loops and use the notation $P(b)$ to indicate the process that event b occurs at. We reach contradiction through a sequence of claims.

Claim: The sequence $P(b_1), P(b_2), \dots$ contains an infinite number of different processes.

If only a finite number of processes appear in the sequence then there is some process P_∞ that appears an infinite number of times in the sequence. But that means that the sequence B contains an infinite number of events at P_∞ all of which are added, by our assumption, by the vacuum loop

at P_∞ . This contradicts our assumption that the vacuum loop completes in a finite number of steps at P_∞ .

Claim: *The sequence B must contain an infinite number of parent/child type pairs $v_{j(J)}(E)^{\text{PR}} \preceq J_{\text{RUN}}$.*

If not, then we can remove an initial segment of B so that it does not contain any such pair. This would leave us only with consecutive pairs that occur at the same process or pairs of the type $k^{\text{QU}} \prec k^{\text{PR}}$. It follows from Lemma 4 that the vacuum loop only creates k^{PR} events for downstream packets. It follows that for all i , $r(P(b_{i+1})) \leq r(P(b_i))$. As a result the sequence B can only involve a finite number of processes, contradicting the first claim.

Claim: *If B contains a pair $k^{\text{QU}} \prec k^{\text{PR}}$, then k is a packet on a self-channel.*

Suppose that B contains an event $b_i = k^{\text{PR}}$ where k is not on a self-channel. It follows from the previous claim that there is a parent/child pair later on in the sequence. Therefore there is a parent/child pair E/J and a segment in B of the form

$$k^{\text{PR}} \prec f_1 \prec f_2 \prec \dots \prec f_N \prec v_{j(J)}(E)^{\text{PR}}$$

where $N \geq 0$; all the events occur at E ; and k is not on the self-channel \overrightarrow{EE} . Because k is not on the self-channel, the event k^{PR} must be generated by step (4) of the vacuum loop, which means that it is generated while $v = r(E) - 1$. This means that the next (and last) notification event that the vacuum loop creates at E is E_{HLT} and not $v_{j(J)}(E)^{\text{PR}}$.

Now we are ready to draw a contradiction. The sequence B contains an infinite number of parent/child pairs and all other pairs in B are local to a process. Therefore there is a segment in B of the form

$$v_{j(J)}(E)^{\text{PR}} \preceq J_{\text{RUN}} \prec f_1 \prec f_2 \prec \dots \prec f_N \prec v_{j(K)}(J)^{\text{PR}} \preceq K_{\text{RUN}}$$

where $N \geq 0$; E/J and J/K are parent/child pairs; and all the events f_1, f_2, \dots, f_N occur at J . It follows that the process J is uninitialized in H_1 , since the J_{RUN} event is generated by the vacuum loop at J . By the Conforming Parent Axiom the process J cannot have a child process K . Contradiction.

We are not done yet. We have constructed a conforming, lossless history H_2 that is fault equivalent to H_1 and which has very good properties. H_2 has no uninitialized processes and according to Corollary 4 all the notifications and APP message broadcast requests in H_2 are processed to completion. But H_2 is not transactional. The reason is that the vacuum loop at each process may create unprocessed packets in the receive queues of non-self downstream channels. Since we generate H_2 from H_1 using a single pass over all the halting processes, these packets may never be revisited.

To make sure that we deal with these packets, we perform a second pass, the same way we performed the first pass. As we have already shown, this procedure creates a conforming, lossless history H_3 that is fault equivalent to H_2 . However unlike the previous pass, all the processes in H_2 have already processed all of their notifications. Therefore for every downstream non-self channel \overrightarrow{PQ} the process P already knows that Q is removed. It now follows from the Process Liveness Axiom that P does not queue any new packets to the channel during the vacuum loop. As a result H_3 is transactional. \square

3 The CBCAST Algorithm

3.1 Introduction

The algorithm we present here is based on the outline in [6]. Our description is more detailed than the authors', but the algorithm itself is a bare bones version of the original. Our prime motivation here is to create a description that is easy to check for correctness. Therefore there is no attempt to accommodate frills like allowing multiple clusters, or optimizing time, space or communication complexity. For such considerations refer back to [6].

From this point on we consider our model to be conforming and so all the histories that we analyze are assumed to be conforming. The properties of CBCAST as presented here do not necessarily hold for non-conforming models.

3.2 Terminology

3.2.1 Messages and delivery

As mentioned in the introduction, we use the term *messages* to refer to the objects being broadcast between processes with the expectation of virtually synchronous delivery. The CBCAST algorithm implements these broadcasts using the underlying multicast of point-to-point packets. When a packet is dequeued at a process and found to contain a message, the message is not immediately delivered in order to preserve causality constraints. Following [6], we say that the message is *received* once the process dequeues its packet up from the receive queue of the channel, but the message within it is *delivered* only at the moment when doing so is consistent with causality constraints. The act of delivery is implemented by invoking the ApplyMessage callback, which applies the message to the user's replicated data (see 2.3.3 below). Every message carries with it three pieces of metadata, denoted $\text{ORIG}(\text{msg})$, $\text{VIEW}(\text{msg})$ and $\text{VT}(\text{msg})$ and describing the originator, view and vector time of the message respectively. These notions are explained below.

3.2.2 Views, installations and view gaps

The membership service sends a coherent stream of membership change notifications to member processes, as described in the previous section. This creates a natural sequence of membership *views*, starting at $\text{view}(0)$ and progressing through $\text{view}(1)$, $\text{view}(2)$, etc. Each view is a finite set of processes, and $\text{view}(n)$ is computed from $\text{view}(n - 1)$ by taking the n^{th} notification of the membership service and applying it to the earlier view, either by adding or removing a process. Each process keeps track of the view notifications as they arrive from the membership service, and then attempts to *install* them. Installation involves waiting a while for packets in flight to arrive at their destinations. This flushing procedure is at the heart of the algorithm and is necessary in order to guarantee virtually synchronous delivery across views. As a result of this wait, a process may be several views behind as old installations are delayed and new view notifications keep arriving. This gap is referred to as the *view gap*. If a process has received a notification of a new view, we

say that the process is *aware* of the view, regardless of whether the process has already installed the view.

It should be noted that a process that leaves the group never re-joins. Even if a user re-starts a process, from the point of view of the membership service the re-started process is brand new.

3.2.3 Instability and forwarding

A major difficulty in designing a coherent broadcast protocol is that it must make broadcasts look like atomic operations, where in reality message packets are sent to each target process individually and are received (or fail to be received) individually. Creating the perception of atomicity requires careful bookkeeping of the progress of each message by the sender and by each target.

In order to broadcast a message to a set of members of the currently installed view, a sender process creates, for each target, a packet containing the message and then sends the packet through the appropriate channel. The sender process tries to keep track of the arrival of the packets. To do that it creates a set, called the *instability set*, that initially contains the identities of all the target processes. When the sender receives an acknowledgment of receipt from a target², it removes that target from the instability set of the message. Likewise, if the group membership service notifies the sender that a target has been removed from the group, that target is removed from the instability set. A message with a non-empty instability set is called *unstable*. The sender keeps copies of all the unstable messages in a *wait set*. If the instability set becomes empty, the message is said to have become *stable* and is removed from the wait set.

When a process receives a packet containing a message, it keeps a copy of the message in a *receive set* where it waits to be delivered. In addition the receiver has a responsibility to help the sender propagate the message to its intended target set. For that purpose the receiver keeps a copy of the received message in a *forwarding queue*. If the receiver learns of the removal of the sender it takes over its duties by re-broadcasting all the messages in the forwarding queue that were received from that sender.

In a practical implementation the receiver tries to keep track of the stabilization of each received message, just like the sender does. Once a message stabilizes there is no more need to help propagate it, even if its sender is removed. Good bookkeeping is essential for keeping forwarding sets small and communication costs low. In our simplified implementation we do not keep track of stabilization on the receiver side, except for the very rudimentary measure of removing obsolete messages from the forwarding queues.

Due to this forwarding mechanism we must differentiate between the *originator* of a message, which is the process that originally broadcast a message, and the packet *sender* which is the process that happened to send the packet containing the message. Usually these two are the same, but in the presence of process removals, a message may be carried from originator to target through a series of forwarded packets. The forwarding procedure naturally leads to duplicated deliveries. The receiving process removes the duplicates using the vector time (see below). There are simple ways to reduce

²In our implementation, acknowledgments are received directly from the target - one acknowledgment per packet. In more practical implementations with lower communication costs, fewer acknowledgments are used and the sender may have other, more indirect methods of deducing that a packet has been received.

the number of duplicates and thus reduce the communication cost that is involved in forwarding. We do not include these here for the sake of simplicity.

3.2.4 Vector time

Each process keeps track of causality relations between messages using a vector of natural numbers, indexed by member processes, called the *vector time*. At each coordinate, the vector time contains the serial number of the latest delivered message that originated at the process that corresponds to that coordinate. The vector time is reset to zero every time a new view is installed. When a message is originally broadcast by a process the vector time is incremented (at the originator's own coordinate) and the metadata of the message - namely $\text{ORIG}(\text{msg})$, $\text{VIEW}(\text{msg})$ and $\text{VT}(\text{msg})$ - are then set to the id of the process, the currently installed view and current vector time of the process, respectively. These values remain fixed for the lifetime of the message. For a detailed discussion, including proofs, of how the vector time is used to guarantee causality order preservation within each view, see [6].

3.2.5 Cluster Initialization and original processes

Birman et al ([6]) assume that the cluster starts at view 1 with a single member. We have to relax this assumption because a central tool in our analysis of CBCAST is the History Reduction Mapping (see Section 5) that moves process joins back to the initial view. Therefore we allow an arbitrary finite number of processes to belong to that view (view zero in our exposition). We call these the *original* processes. These processes get started through an invocation of the `protStart` procedure. We assume that the procedure gets called at each original process at exactly the same time and with the same roster of members that includes exactly the set of original processes. While the assumption of simultaneity is not realistic, we only need to use it with theoretical "reduced" cases and not with actual clusters which can still be assumed to start with a single member.

3.3 Outline of the Algorithm

Each process, from the moment it joins the group to the moment it leaves, keeps track of the views as notifications are received from the membership service. For each view from the currently installed view to the most recently announced view, the process keeps a list of the members of that view.

Usually, when the process needs to broadcast a message to the group, it fixes the metadata of the message with the current view and vector time, and then sends a packet containing the message to each member of the current view. The process also places the message in a wait set, where it tracks its stabilization as the recipient processes acknowledge the message. However, when there is a view gap (i.e. there are announced views that have not been installed yet) the process refrains from broadcasting messages or fixing their metadata, and it queues them instead. The messages are broadcast whenever the view gap closes.

When a process receives a packet containing a message, it acknowledges its receipt to the sender (the sender, remember, may be different from the originator). If the message is not a duplicate it

is placed in the receive set until it becomes deliverable and a copy of it is created and appended to the tail of the forwarding queue of the sender of the message. When a message becomes deliverable it is removed from the receive set and applied to the user's replicated data. At the time of delivery the process updates its own vector time by incrementing the coordinate that corresponds to the originator of the message.

When a process is notified that another member process has been removed, it takes each message in the forwarding queue of that process and forwards it to all the live processes. These messages are forwarded with their original metadata (originator, view and vector time) unchanged. A copy of each message is placed in the wait set, to await stabilization.

Whenever a view gap exists, such as after a new view notification is received, the process must wait for its wait set to empty out before it can install the next view. Once the wait set becomes empty, the process sends a *flush* packet to all the live processes. This packet contains the value of the latest view known to the process (i.e. the current view plus the view gap). The process then waits to receive similar flush packets from all the live processes. Once that happens, the process installs the next view. It applies a view installation notification to the user's replicated data and removes any obsolete messages from the receive set and the forwarding queue.

Our implementation contains, in addition to flush packets, a related type of packet called a *ghost* packet. These packets are not necessary in a practical implementation. We use them to facilitate our reasoning about joining processes. A ghost packet is the "ghost" of a flush packet that would have been sent by a child of an existing process, had that child already been born.

When a new process joins the group, it must somehow synchronize its state with the state of the existing processes. This is done in two stages. Initially the new process starts life as a perfect replica of an existing process, the *parent*. We do not describe how this is done, and subsume it into the opaque membership service. In addition the new process must compensate for the natural race conditions that occur as a result of the fact that packets have been in flight between its parent and the other members of the group at the moment that it is born. This compensation is performed using the *donation* protocol, whereby each existing process other than the parent exchanges instability information with the new process.

3.4 Variable and Function Definitions

3.4.1 Global variables - the state of a process

cur_view The number of the current view.

v_gap The number of yet-uninstalled views of which we have been notified by the membership service.

self Local process identifier.

MSet The set of identifiers of the member processes of the current view.

PendViewQueue A queue of pending view changes. Each view change is either a joining of a new process or the removal of an existing process.

LiveSet The set of identifiers of all the live processes. This includes every process that is a member of the current view or a known future view, excluding all known removed processes.

ContactSet A subset of **LiveSet** that excludes all the processes that joined before the local process but from which a donation has not yet been received.

vt[] A vector of natural numbers, indexed by the process identifiers of the members of the current view. This is the vector time of the local process.

ReceiveSet The set of non-duplicate messages that were received and not yet discarded or delivered.

FwdQueue[] A vector of message queues, indexed by process identifiers. For each process identifier, the queue contains copies of all the messages that were sent from and acknowledged to that process - excluding duplicates - in the order they were received. The queue includes messages that were merely forwarded by the sending process and did not originate from it. Each queue includes both delivered and undelivered messages.

WaitSet The set of all the messages that the process broadcast or forwarded during the current view (note that forwarded messages may have a **VIEW(msg)** of a higher view even if they are forwarded during the current view). Each message in the wait set is paired with an index and an instability set. The index indicates how many messages were broadcast or forwarded out of the process prior to the current message. The instability set contains, for every process that has not yet acknowledged the message, an index that indicates how many broadcast and forwarded messages were received from that process prior to the broadcasting or forwarding of the current message. **WaitSet** is organized as the union of two data structures:

1. **BcastWaitSet** contains only the messages that were broadcast by the current process.
2. **FwdWaitSet** contains only the messages that were forwarded by the current process.

LaunchQueue A queue of all the unsent messages that need to be broadcast once the view gap closes.

ReplicatedData An opaque object containing the replicated user data. This data is managed by the user application in an application-specific way, subject to the rules listed in subsection 2.3.3.

ghost_height A number indicating the highest ghost value sent by the process so far. Ghost values are sent out in a strictly increasing sequence.

flush_height A number indicating the highest flush value sent by the process so far. Flush values are sent out in a strictly increasing sequence.

ghost[] A vector of view numbers, indexed by process identifiers. It keeps, for each process, the highest ghost value that was received from that process.

flush[] A vector of view numbers, indexed by process identifiers. It keeps, for each process, the highest flush value that was received from that process.

mpkt_out A counter of outbound messages, made up of two fields:

- *mpkt_out.b* is the number of original messages broadcast by the process up to this point.
- *mpkt_out.f* is the number of messages forwarded by the process up to this point.

$mpkt_in[]$ A vector of number pairs, indexed by process identifiers, that counts how many messages have been received from each process so far. The two fields are:

- $mpkt_in[P].b$ is the number of original P -messages received from P .
- $mpkt_in[P].f$ is the number of forwarded messages received from P

3.4.2 Packet and notification types

$n_{JOIN}(\mathbf{pid}, \mathbf{p_pid})$ Notification that a new process with identifier \mathbf{pid} joined the group as a clone of the parent process with identifier $\mathbf{p_pid}$.

$n_{REM}(\mathbf{pid})$ Notification that current member process with identifier \mathbf{pid} was removed from the group.

$p_{MSG}(\mathbf{msg})$ A message packet carrying a message \mathbf{msg} .

$p_{ACK}(\mathbf{msg})$ An acknowledgement packet carrying an acknowledgment of receipt of message \mathbf{msg} .

$p_{GHOST}(\mathbf{v})$ A ghost packet indicating ghost value \mathbf{v} .

$p_{FLUSH}(\mathbf{v})$ A flush packet indicating flush up to view \mathbf{v} .

$p_{GHOST}(\geq \mathbf{v})$ or $p_{FLUSH}(\geq \mathbf{v})$ Stand for any packet $p_{GHOST}(\mathbf{v}')$ or $p_{FLUSH}(\mathbf{v}')$ where $\mathbf{v}' \geq \mathbf{v}$.

$p_{DONATE}(\mathbf{donation})$ A donation packet containing a donation of instability information from an existing process to a newly joined process.

$p_{CO-DONATE}(\mathbf{co-donation})$ A co-donation packet containing a donation of instability information from a newly joined process to an existing process.

3.4.3 Message metadata

Each message is fixed with three pieces of metadata

$ORIG(\mathbf{msg})$ Originator, namely the process that broadcast the message originally.

$VIEW(\mathbf{msg})$ View, which is the view of the originator when the message is broadcast.

$VT(\mathbf{msg})$ Vector Time, which is (roughly) the vector time of the originator when the message is broadcast.

The pair $\langle VIEW(\mathbf{msg}), VT(\mathbf{msg}) \rangle$ uniquely identifies the message \mathbf{msg} .

3.5 Detailed CBCAST Algorithm Pseudo Code

Our pseudo code implements the various PROTOCOL interfaces (see 2.3.2)

protBroadcast(m) at page 44
 protStart(roster, P) at page 45
 protRun(P) at page 46
 protRemove(P) at page 47
 protJoin(P, E) at page 48
 protPacket(k, S) at page 49

The protPacket(k, S) interface implementation uses the following procedures to process the various types of packets that are defined in the CBCAST protocol:

ReceiveMessage(msg, sender) at page 50
 ReceiveAck(msg, sender) at page 50
 ReceiveGhost(view, sender) at page 51
 ReceiveFlush(view, sender) at page 51
 ReceiveDonation(donation, sender) at page 52
 ReceiveCoDonation(co_donation, sender) at page 53

In addition there are three service routines that are called from several places that deal with view installation and message delivery:

CheckFlush() at page 54
 TryToInstall() at page 55
 Scan() at page 56

Interface protBroadcast(msg)

Input: msg is the message that is being broadcast

```

if  $v\_gap > 0$  then
1   append msg to the tail of LaunchQueue;
end
else
   increment  $mpkt\_out.b$ ;
   // calculate the message vector time
   let  $vt' = vt$ ;
   let  $vt'[self] = vt[self] + mpkt\_out.b - mpkt\_in[self].b$ ;
   // fix message metadata before broadcasting
    $ORIG(msg) \leftarrow self$ ;
    $VIEW(msg) \leftarrow cur\_view$ ;
    $VT(msg) \leftarrow vt'$ ;
2   queue  $p_{MSG}(msg)$  to ContactSet; // multicast the message packets
   let  $index = mpkt\_out$ ; // locates msg in the outgoing message sequence
   let  $iset[] = mpkt\_in[]$ ; // the initial instability set for msg
   add  $\langle msg, index, iset[] \rangle$  to BcastWaitSet;
end

```

Interface protStart(roster, pid)

Input: roster is the set of original members of the group (view zero members). pid is the process identifier of the local process

```
GroundState(); // create the initial value of ReplicatedData
let cur_view = 0;
let v_gap = 0;
let self = pid;
let MSet = roster;
let PendViewQueue =  $\emptyset$ ;
let LiveSet = ContactSet = roster;
let vt =  $\emptyset$ ;
let ReceiveSet =  $\emptyset$ ;
let FwdQueue =  $\emptyset$ ;
let WaitSet =  $\emptyset$ ;
let LaunchQueue =  $\emptyset$ ;
let ghost_height = flush_height = 0;
let ghost[] = flush[] =  $\emptyset$ ;
let mpkt_out = {f = 0; b = 0};
let mpkt_in[] =  $\emptyset$ ;
foreach id  $\in$  roster do
    create vt[id] = 0;
    create FwdQueue[id] =  $\emptyset$ ;
    create mpkt_in[id] = {f = 0; b = 0};
    create ghost[id] = flush[id] = 0;
    ApplyJoin(id);
end
// We launch the main APP thread asynchronously
// It will start executing at some indeterminate point in the future
execute Main(self);
```

Interface `protRun(pid)`

Input: `pid` is the process identifier of the new process

```
increment v-gap;  
append  $\langle \text{JOIN}, \text{pid} \rangle$  to the tail of PendViewQueue;  
add pid to LiveSet;  
let ContactSet =  $\{\text{pid}\}$ ;  
create FwdQueue[pid] =  $\emptyset$ ;  
let BcastWaitSet =  $\emptyset$ ;  
foreach  $\langle \text{msg}, \text{index}, \text{iset}[] \rangle \in \text{FwdWaitSet}$  do  
    let index.b = 0;  
end  
let LaunchQueue =  $\emptyset$ ;  
let flush.height = ghost.height;  
create ghost[pid] = ghost.height;  
create flush[pid] = ghost.height;  
let mpkt_out.b = 0;  
create mpkt_in[pid] = mpkt_out;  
let self = pid;  
CheckFlush();
```

Interface `protRemove(rem_proc)`

Input: `rem_proc` is the identifier of the removed process

```
1  increment v-gap;
2  append  $\langle \text{REMOVE}, \text{rem\_proc} \rangle$  to the tail of PendViewQueue;
   remove rem_proc from LiveSet;
   remove rem_proc from ContactSet;
   foreach  $\langle \text{msg}, \text{index}, \text{iset} \rangle \in \text{WaitSet}$  do
       discard iset[rem_proc];
       if iset =  $\emptyset$  then
           remove  $\langle \text{msg}, \text{index}, \text{iset} \rangle$  from WaitSet; // message is stable
       end
   end
   discard mpkt_in[rem_proc];
3  while FwdQueue[rem_proc]  $\neq \emptyset$  do
       pop msg from the head of FwdQueue[rem_proc];
       // create an instability set that contains
       // all the live processes
       increment mpkt_out.f;
4       let index = mpkt_out;
       let iset[] = mpkt_in[];
5       queue pMSG  $\langle \text{msg} \rangle$  to ContactSet; // multicast the message packets
6       add  $\langle \text{msg}, \text{index}, \text{iset}[] \rangle$  to FwdWaitSet;
   end
   discard FwdQueue[rem_proc];
   discard ghost[rem_proc];
   discard flush[rem_proc];
   CheckFlush();
```

Interface protJoin(jn_proc, p_proc)

Input: *jn_proc* is the identifier of the joining process. *p_proc* is the identifier of the parent process.

```
increment v_gap;
append ⟨JOIN, jn_proc⟩ to the tail of PendViewQueue;
add jn_proc to LiveSet;
add jn_proc to ContactSet;
create FwdQueue[jn_proc] = ∅;
foreach ⟨msg, index, iset[]⟩ ∈ WaitSet do
  if iset[p_proc] exists then
1      create iset[jn_proc] = {f = iset[p_proc].f; b = 0};
  end
end
create ghost[jn_proc] = ghost[p_proc];
create flush[jn_proc] = ghost[p_proc]; // The received flush value of the new process
is inherited from the received ghost height of the parent
create mpkt_in[jn_proc] = {f = mpkt_in[p_proc].f; b = 0};
let donation = ⟨WaitSet, mpkt_in[], ghost_height, flush_height⟩;
queue pDONATE⟨donation⟩ to jn_proc;
CheckFlush();
```

Interface `protPacket(k , sender)`

Input: k is the packet being received. `sender` is the process identifier of the sender of the packet

```
switch cont( $k$ ) do
  case  $p_{\text{MSG}}$   $\langle \text{msg} \rangle$ :
    ReceiveMessage(msg, sender);
  endsw
  case  $p_{\text{ACK}}$   $\langle \text{msg} \rangle$ :
    ReceiveAck(msg, sender);
  endsw
  case  $p_{\text{GHOST}}$   $\langle \text{view} \rangle$ :
    ReceiveGhost(view, sender);
  endsw
  case  $p_{\text{FLUSH}}$   $\langle \text{view} \rangle$ :
    ReceiveFlush(view, sender);
  endsw
  case  $p_{\text{DONATE}}$   $\langle \text{donation} \rangle$ :
    ReceiveDonation(donation, sender);
  endsw
  case  $p_{\text{CO-DONATE}}$   $\langle \text{co\_donation} \rangle$ :
    ReceiveCoDonation(co_donation, sender);
  endsw
endsw
```

Procedure ReceiveMessage(msg, sender)

Input: msg is the message being received. sender is the process identifier of the sender of the message packet

```
    queue pACK(msg) to sender; // acknowledge receipt of the message
    if ORIG(msg) = sender then
        increment mpktin[sender].b;
    end
    else
        increment mpktin[sender].f;
    end
1  // Check for duplicates:
    if VIEW(msg) < curview then
2      discard pMSG(msg); // obsolete messages are duplicates (Lemma 34)
    end
    else if VIEW(msg) = curview and vt[ORIG(msg)] ≥ VT(msg)[ORIG(msg)] then
        discard pMSG(msg); // duplicate - message already delivered
    end
    else if msg ∈ ReceiveSet then
        discard pMSG(msg); // duplicate - message already received
    end
    else
3      add msg to ReceiveSet;
4      append msg to the tail of FwdQueue[sender];
5      Scan(); // scan ReceiveSet and deliver all the deliverable messages
    end
end
```

Procedure ReceiveAck(msg, sender)

Input: msg is the message being acknowledged. sender is the process identifier of the sender of the acknowledgement packet

```
// the following if statement will always succeed
if ⟨msg, index, iset⟩ ∈ WaitSet exists then
    discard iset[sender];
    if iset = ∅ then
        remove ⟨msg, index, iset⟩ from WaitSet; // message is stable
        CheckFlush();
    end
end
end
```

Procedure ReceiveGhost(view, sender)

Input: view is the ghost height of the sender. sender is the process identifier of the sender of the packet

let *ghost*[sender] = view; // *ghost*[sender] always increases

Procedure ReceiveFlush(view, sender)

Input: view is the flush height of the sender. sender is the process identifier of the sender of the packet

let *flush*[sender] = view; // *flush*[sender] always increases
TryToInstall();

Procedure ReceiveDonation(donation, sender)

Input: donation is the donation being received. sender is the process identifier of the sender of the donation packet

```
add sender to ContactSet;
let co_donation =  $\langle \text{WaitSet}, \text{mpkt\_in}[], \text{ghost\_height}, \text{flush\_height} \rangle$ ;
queue  $\mathbf{p}_{\text{CO-DONATE}}$ (co_donation) to sender; // Co-donate local state to the sender
// Process, in order, all the untimely packets
let  $\text{UNT}_g = \{ \langle \text{msg}, \text{index}, \text{iset}[] \rangle \in \text{WaitSet} \mid \text{iset}[\text{sender}] \text{ exists} \}$ ;
Define  $\text{height}_1(\langle \text{msg}, \text{index}, \text{iset}[] \rangle \in \text{UNT}_g) = \text{index}.b + \text{index}.f$ ;
Define  $\text{height}_2(\langle \text{msg}, \text{index}, \text{iset}[] \rangle \in \text{UNT}_g) = 0$ ;
let  $\text{UNT}_p = \{ \langle \text{msg}, \text{index}, \text{iset}[] \rangle \in \text{donation.WaitSet} \mid \text{iset}[\text{self}] \text{ exists} \}$ ;
Define  $\text{height}_1(\langle \text{msg}, \text{index}, \text{iset}[] \rangle \in \text{UNT}_p) = \text{iset}[\text{self}].b + \text{iset}[\text{self}].f$ ;
Define  $\text{height}_2(\langle \text{msg}, \text{index}, \text{iset}[] \rangle \in \text{UNT}_p) = \text{index}.b + \text{index}.f$ ;
let  $\text{UNT} = \text{UNT}_g \cup \text{UNT}_p$ ;
sort UNT using the lexicographical order ( $\text{height}_1, \text{height}_2$ );
// we process the elements of UNT in order
foreach  $\langle \text{msg}, \text{index}, \text{iset}[] \rangle \in \text{UNT}$  do
  if  $\langle \text{msg}, \text{index}, \text{iset}[] \rangle \in \text{UNT}_p$  then
    if  $\text{index}.b + \text{index}.f > \text{mpkt\_in}[\text{sender}].b + \text{mpkt\_in}[\text{sender}].f$  then
      // we found an untimely message packet from the sender to the parent,
      and we process its clone now
1      ReceiveMessage(msg, sender);
    end
  end
  if  $\langle \text{msg}, \text{index}, \text{iset}[] \rangle \in \text{UNT}_g$  then
    if  $\text{index}.b + \text{index}.f \leq \text{donation.mpkt\_in}[\text{self}].b + \text{donation.mpkt\_in}[\text{self}].f$  then
      // we found a message packet from the parent whose acknowledgement
      packet was untimely, so we process its clone now
2      ReceiveAck(msg, sender);
    end
  end
end
let  $\text{ghost}[\text{sender}] = \text{donation.ghost\_height}$ ;
let  $\text{flush}[\text{sender}] = \text{donation.flush\_height}$ ;
```

Procedure ReceiveCoDonation(co_donation, sender)

Input: co_donation is the co-donation being received. sender is the process identifier of the sender of the co-donation packet

```
// Process, in order, all the untimely and post-critical packets
let UNTg = {⟨msg, index, iset[]⟩ ∈ co_donation.WaitSet | iset[self] exists};
Define height1(⟨msg, index, iset[]⟩ ∈ UNTg) = iset[self].b + iset[self].f;
Define height2(⟨msg, index, iset[]⟩ ∈ UNTg) = index.b + index.f;
let UNTp = {⟨msg, index, iset[]⟩ ∈ WaitSet | iset[sender] exists};
Define height1(⟨msg, index, iset[]⟩ ∈ UNTp) = index.b + index.f;
Define height2(⟨msg, index, iset[]⟩ ∈ UNTp) = 0;
let UNT = UNTg ∪ UNTp;
sort UNT using the lexicographical order (height1, height2);
// we process the elements of UNT in order
foreach ⟨msg, index, iset[]⟩ ∈ UNT do
  if ⟨msg, index, iset[]⟩ ∈ UNTg then
    if index.b + index.f > mpkt_in[sender].b + mpkt_in[sender].f then
      // we found one of two things here:
      // either an untimely forwarded message packet from the parent that
      // we process now as a message from the sender
      // or a post-critical, pre-donation forwarded message packet from the
      // sender that we process now
1      ReceiveMessage(msg, sender);
    end
  end
  if ⟨msg, index, iset[]⟩ ∈ UNTp then
    if index.b + index.f ≤ co_donation.mpkt_in[self].b + co_donation.mpkt_in[self].f then
      // we found a timely message packet from us to the parent whose
      // acknowledgement was untimely, so we process its clone now
2      ReceiveAck(msg, sender);
    end
  end
end
let ghost[sender] = co_donation.ghost_height;
let flush[sender] = co_donation.flush_height;
3 TryToInstall();
```

Procedure CheckFlush

```
    if FwdWaitSet  $\neq \emptyset$  then
        return; // there are unstable forwarded messages, do not send any ghosts or
        flushes
    end
    if ghost_height < cur_view + v_gap then
        let ghost_height = cur_view + v_gap;
1      queue  $p_{\text{GHOST}}$ (ghost_height) to ContactSet; // multicast the ghost packets
    end
    if BcastWaitSet  $\neq \emptyset$  then
        return; // there are unstable original messages, do not send any flushes
    end
    if flush_height < cur_view + v_gap then
        let flush_height = cur_view + v_gap;
2      queue  $p_{\text{FLUSH}}$ (flush_height) to ContactSet; // multicast the flush packets
    end
```

Procedure TryToInstall

```
    // check whether all the members are fully flushed
1  foreach pid ∈ LiveSet do
    if flush[pid] < cur_view + v_gap then
        return ; // some members are not flushed - wait
    end
end
while v_gap > 0 do    // this loop installs all the pending views
2  // remove obsolete messages from ReceiveSet
    foreach msg ∈ ReceiveSet do
        if VIEW(msg) = cur_view then
            remove msg from ReceiveSet;
        end
    end
3  // remove obsolete messages from FwdQueue
    foreach pid ∈ LiveSet do
        foreach msg ∈ FwdQueue[pid] do
            if VIEW(msg) = cur_view then
                remove msg from FwdQueue[pid];
            end
        end
    end
    increment cur_view;
    decrement v_gap;
    pop notification from the head of PendViewQueue;
    if notification = ⟨JOIN, pid⟩ then
        add pid to MSet;
        ApplyJoin(pid); // deliver notification to APP
        if pid = self then
            execute Main(self); // launch the main APP thread asynchronously
        end
    end
    else if notification = ⟨REMOVE, pid⟩ then
        remove pid from MSet;
        ApplyRemoval(pid); // deliver notification to APP
    end
4  reset vt; // New view now installed.  vt coordinates reflect new membership
    Scan(); // high view messages may now become deliverable
end
// v_gap = 0 - time to broadcast all pending messages
while LaunchQueue ≠ [] do
    pop msg from the head of LaunchQueue;
5  protBroadcast(msg);
end
```

Procedure Scan

```
// look for deliverable messages. A message is deliverable if all the
// following are true:
// 1. It is a current-view message
// 2. It is the next expected message from its originator
// 3. All the messages on which it depends have been delivered already
let deliverable_messages_found = false;
foreach msg ∈ ReceiveSet do
  if VIEW(msg) = cur_view then
    // msg is a current-view message
    if VT(msg)[ORIG(msg)] = vt[ORIG(msg)] + 1 then
      // msg is the next expected message from its originator
      let all_dependents_delivered = true;
      foreach pid ∈ MSet and pid ≠ ORIG(msg) do
        if VT(msg)[pid] > vt[pid] then
          let all_dependents_delivered = false;
        end
      end
      if all_dependents_delivered = true then
        // msg is deliverable
        let deliverable_messages_found = true;
        increment vt[ORIG(msg)];
        remove msg from ReceiveSet;
        let originator = ORIG(msg);
        strip out metadata stamps VIEW(msg), VT(msg) and ORIG(msg);
        ApplyMessage(msg, originator); // deliver message to APP
      end
    end
  end
end
if deliverable_messages_found = true then
  Scan(); // try to see if more messages can now be delivered
end
```

4 Basic Properties Of The CBCAST Algorithm

In subsequent sections we will analyze the CBCAST protocol in depth. Right now we want to highlight some of its important basic properties.

4.1 Some CBCAST invariants

Definition 16.

- Let T be a transaction. The trigger of T is denoted $\text{trig}(T)$
- Let $e \in \mathbb{E}_P$. Then e belongs to a unique transaction T . We denote $T = \text{trans}(e)$ and use $\text{trig}(e)$ as shorthand for $\text{trig}(\text{trans}(e))$.
- Let T be a transaction. The **view** of T is $\text{view}(\text{trig}(T))$ and denoted by $\text{view}(T)$. Since the side effects of T cannot contain notification events, all the events in T share the same view $\text{view}(T)$.

Definition 17. Let P be any process in a group that executes the CBCAST protocol. Let var be any state variable (see 3.4.1) and let $e \in \mathbb{E}_P$ be any event other than the join event of P , in other words $e \neq v_{j(P)}(P)^{\text{PR}}$.

If e is a trigger event we use the notation $\text{var}_{P@e}$ to denote the value of the variable var at process P at the onset of the transaction $\text{trans}(e)$. We use the notation $\text{var}^{P@e}$ to denote the value of the variable var at process P at the conclusion of the transaction $\text{trans}(e)$.

If e is a queuing event we use the notation $\text{var}_{P@e}$ to denote the value of the variable var at process P at the moment when the queuing event occurs. Since queuing events do not change state variables, there is no distinction here between the pre and post values.

When $e = v_{j(P)}(P)^{\text{PR}}$, the processing of e causes the execution of the protStart procedure or the protRun procedure, according as P is an original process (a member of view zero) or a late joining process. We use the same definition of $\text{var}^{P@e}$ that we use for any other trigger. However for an original process we define

$$\text{var}_{P@e} = \text{var}^{P@e}$$

and for a late joining process we define $\text{var}_{P@e}$ to be the value of var right before the invocation of the CheckFlush procedure at the end of the protRun procedure.

This last part of the definition is admittedly not elegant. However it does have some intuitive justification in the sense that the endpoint of protStart and the pre- CheckFlush point in the protRun procedure are the first points in the life of a process where it is fully initialized as a CBCAST process.

Definition 18. Let P be a process and let $e \in \mathbb{E}_P$ be any trigger event. Let \mathbb{U}_e be the set of processes that had not yet contacted P at the time that e occurred. These are processes that joined the group before P did, but for which P has not yet processed a donation packet. Formally

$$\mathbb{U}_e = \{Q \in \mathbb{P} \parallel j(Q) < j(P) \text{ and if } d = \mathbf{p}_{\text{DONATE}} \langle \rangle \in \overrightarrow{QP} \text{ then either } d^{\text{PR}} \succ e \text{ or } d^{\text{PR}} \text{ does not exist}\}$$

In particular if P is a member of view zero then $\mathbb{U}_e = \emptyset$.

The set \mathbb{U}_e is called the **uncontacted set** of e .

Lemma 8 (CBCAST Omnibus Lemma).

Let P be any process and let e be any trigger event in P . Then the following relations hold at P :

1. $(\text{cur_view} + \text{v_gap})^{P@e} = \text{view}(e)$
- 2.

$$\begin{aligned} \text{LiveSet}^{P@e} &= \{Q \in \mathbb{P} \mid j(Q) \leq (\text{cur_view} + \text{v_gap})^{P@e} < r(Q)\} \\ \text{ContactSet}^{P@e} &= \text{LiveSet}^{P@e} \setminus \mathbb{U}_e \end{aligned}$$

3. The entries in the vectors $\text{FwdQueue}[]^{P@e}$, $\text{mpkt_in}[]^{P@e}$, $\text{ghost}[]^{P@e}$ and $\text{flush}[]^{P@e}$ correspond exactly to the members of $\text{LiveSet}^{P@e}$.
4. The entries in the vector $\text{vt}[]^{P@e}$ correspond exactly to the members of $\text{MSet}^{P@e}$.
5. For any $X \in \text{LiveSet}^{P@e}$

$$\text{flush}[X]^{P@e} \leq \text{ghost}[X]^{P@e} \leq (\text{cur_view} + \text{v_gap})^{P@e}$$

If $X \notin \text{ContactSet}^{P@e}$ then the right inequality is strict. If $\text{v_gap}^{P@e} = 0$ then the inequalities are actually equalities.

6. For any $X \in \text{LiveSet}_{P@e} \cap \text{LiveSet}^{P@e}$

$$\begin{aligned} \text{ghost}[X]_{P@e} &\leq \text{ghost}[X]^{P@e} \\ \text{flush}[X]_{P@e} &\leq \text{flush}[X]^{P@e} \end{aligned}$$

in other words the values of $\text{ghost}[X]$ and $\text{flush}[X]$ are non-decreasing.

7. If $e = v_i(P)^{\text{PR}}$ and $X \in \text{LiveSet}^{P@e}$ then

$$\begin{aligned} \text{ghost}[X]^{P@v_i(P)^{\text{PR}}} &\leq \text{ghost_height}_{X@v_i(X)^{\text{PR}}} \\ \text{flush}[X]^{P@v_i(P)^{\text{PR}}} &\leq \text{flush_height}_{X@v_i(X)^{\text{PR}}} \end{aligned}$$

8. $\text{flush}[P]^{P@e} \leq \text{flush_height}^{P@e} \leq \text{ghost_height}^{P@e} \leq (\text{cur_view} + \text{v_gap})^{P@e}$

9. If $\text{v_gap}^{P@e} > 0$ then

$$\begin{aligned} \text{ghost_height}^{P@e} &= (\text{cur_view} + \text{v_gap})^{P@e} \text{ if and only if } \text{FwdWaitSet}^{P@e} = \emptyset \\ \text{flush_height}^{P@e} &= (\text{cur_view} + \text{v_gap})^{P@e} \text{ if and only if } \text{WaitSet}^{P@e} = \emptyset \end{aligned}$$

10. If $\text{v_gap}^{P@e} = 0$ then $\text{LaunchQueue}^{P@e} = \emptyset$.

Proof. The proof proceeds by induction on e , where we assume by induction that the lemma holds for f if either $f \prec e$ or if $\text{view}(f) < \text{view}(e)$. This is possible thanks to Corollary 3 and Lemma 2.

We have three types of triggers to consider: message broadcast request events, notification events and packet dequeuing events. In the latter case we will use the following notation throughout. k is the packet that is being dequeued (so $e = k^{\text{PR}}$) and X is the process that queued the packet k .

We start by picking off the easy cases. We show that if e is a packet dequeuing event or a message broadcast request event then claims (1), (2) and (3) all hold. claim (7) holds vacuously for e because it only pertains to notification events.

For the remaining cases, since every trigger event causes some **CBCAST** procedure to be executed, we simply go over each procedure and show that if the inductive hypothesis is assumed then the lemma holds at the end of the execution of the procedure.

To prove claim (1) when e is a packet dequeuing event we need two facts. First, the proof of Corollary 2 demonstrates that $\text{view}(e) = \text{view}(e')$, where $e' \prec e$ is the immediate predecessor of e in \mathbb{E}_P . Second, a lengthy but routine inspection of the pseudo-code shows that the `protPacket` procedure does not change the sum $\text{cur_view} + \text{v_gap}$ (the `TryToInstall` utility procedure increments cur_view and decrements v_gap zero or more times, but does not change their sum). These facts taken together with the inductive hypothesis give

$$\text{view}(e) = \text{view}(e') = (\text{cur_view} + \text{v_gap})^{P@e'} = (\text{cur_view} + \text{v_gap})_{P@e} = (\text{cur_view} + \text{v_gap})^{P@e}$$

The exact same argument holds when e is a message broadcast request event (with `protBroadcast` replacing `protPacket`).

To prove the first part of claim (2) when e is a packet dequeuing event or a message broadcast request event, notice that neither `protPacket` nor `protBroadcast` change the value of `LiveSet`, and as we already saw these procedures do not change the value of $\text{cur_view} + \text{v_gap}$ either. As a result this part of the claim follows by induction.

An immediate corollary is that if e is a packet dequeuing event then $X \in \text{LiveSet}^{P@e}$. This is because the definition of $\text{view}(e)$ and the Conforming Packet Axiom imply that $j(X) \leq \text{view}(e) < r(X)$. It follows from claim (1) and the first part of claim (2) that $X \in \text{LiveSet}^{P@e}$.

The second part of the claim is a bit more complicated. As long as e is not a donation packet it is easy to check that `ContactSet` does not change and $\mathbb{U}_e = \mathbb{U}_{e'}$, where e' is the immediate predecessor of e in \mathbb{E}_P and so this part of the claim follows by induction.

If e is the donation packet from X then it is easy to see that $X \notin \mathbb{U}_e$ and that $\mathbb{U}_e = \mathbb{U}_{e'} \setminus \{X\}$. We already demonstrated that $X \in \text{LiveSet}^{P@e}$. The `protPacket` procedure executes the `ReceiveDonation` procedure which in turn adds X to `ContactSet` and so by induction

$$\begin{aligned} \text{ContactSet}^{P@e} &= \text{ContactSet}_{P@e} \cup \{X\} = \text{ContactSet}^{P@e'} \cup \{X\} = \\ &= (\text{LiveSet}^{P@e'} \setminus \mathbb{U}_{e'}) \cup \{X\} = (\text{LiveSet}^{P@e} \setminus \mathbb{U}_{e'}) \cup \{X\} = \\ &= (\text{LiveSet}^{P@e} \cup \{X\}) \setminus (\mathbb{U}_{e'} \setminus \{X\}) = \text{LiveSet}^{P@e} \setminus \mathbb{U}_e \end{aligned}$$

Proving claim (3) when e is a packet dequeuing event or a message broadcast request event amounts to a routine check that only notification related procedures, namely `protStart`, `protRun`, `protJoin` and `protRemove` actually add or remove entries from the mentioned vectors or change `LiveSet`.

We prove the remaining claims for non-notification events by examining the `protBroadcast` procedure and each of the service procedures that are invoked by the `protPacket` procedure. A claim has to be tested against a procedure only if the procedure changes one or more of the variables that are mentioned in the claim.

protBroadcast

This procedure only affects claims (9) and (10). If $v_gap = 0$ it adds a record to `WaitSet`, but in that case claim (9) is vacuously true. If $v_gap > 0$ it adds a message to `LaunchQueue`, but in this case claim (10) is vacuously true.

ReceiveMessage

This procedure does not affect any of the claims because it does not change any of the relevant variables (this includes the invocation of the `Scan` procedure which also does not change any of the relevant variables).

ReceiveAck

This procedure affects claims (8) and (9) by making changes to `WaitSet`, `ghost_height` and `flush_height` but no other variable.

Claim (8) is true by induction before `CheckFlush` is called. The claim remains true if `CheckFlush` sets $ghost_height = cur_view + v_gap$. There are two possible impediments to this action. If `FwdWaitSet` $\neq \emptyset$ then `CheckFlush` does nothing and the claim remains true by induction. If `ghost_height` is already high before `CheckFlush` is called the claim remains true regardless of whether `CheckFlush` raises `flush_height` or not. So (8) remains true in all cases.

Claim (9) is vacuously true if $v_gap = 0$ so assume that $v_gap > 0$. The procedure may shrink, but does not enlarge, either `FwdWaitSet` or `BcastWaitSet` and if it removes any record, it invokes the `CheckFlush` procedure. If `WaitSet` does not lose a record then `CheckFlush` is not called and nothing changes. If `WaitSet` loses a record, we have to look at the following cases:

FwdWaitSet remains non-empty after the record loss

In this case `CheckFlush` does nothing and the claim remains true by induction.

FwdWaitSet becomes empty while BcastWaitSet remains non-empty

In this case it follows from the inductive hypothesis that

$$\begin{aligned} ghost_height_{P@e} &< cur_view + v_gap \\ flush_height_{P@e} &< cur_view + v_gap \end{aligned}$$

and therefore `CheckFlush` sets $ghost_height^{P@e} = cur_view + v_gap$ and does not touch `flush_height`. These changes preserve the claims of (9).

FwdWaitSet becomes empty while BcastWaitSet remains empty

In this case it follows from the inductive hypothesis that

$$\begin{aligned} ghost_height_{P@e} &< cur_view + v_gap \\ flush_height_{P@e} &< cur_view + v_gap \end{aligned}$$

and therefore `CheckFlush` sets

$$ghost_height^{P@e} = flush_height^{P@e} = cur_view + v_gap$$

These changes preserve the claims of (9).

FwdWaitSet remains empty while BcastWaitSet remains non-empty

In this case it follows from the inductive hypothesis that

$$\begin{aligned} \text{ghost_height}_{P@e} &= \text{cur_view} + v_gap \\ \text{flush_height}_{P@e} &< \text{cur_view} + v_gap \end{aligned}$$

and therefore CheckFlush does nothing and the claim remains true by induction.

FwdWaitSet remains empty while BcastWaitSet becomes empty

In this case it follows from the inductive hypothesis that

$$\begin{aligned} \text{ghost_height}_{P@e} &= \text{cur_view} + v_gap \\ \text{flush_height}_{P@e} &< \text{cur_view} + v_gap \end{aligned}$$

and therefore CheckFlush sets $\text{flush_height}^{P@e} = \text{cur_view} + v_gap$ and does not touch ghost_height . These changes preserve the claims of (9).

ReceiveGhost

This procedure affects claims (5) and (6), but only with respect to the sender process X . We know that $\text{LiveSet}_{P@e} = \text{LiveSet}^{P@e}$ and that $X \in \text{LiveSet}^{P@e}$ so both claims must be verified for X .

Also note that it follows from claim (3) that for every process in $\text{LiveSet}^{P@e}$ all the fields in claims (5) and (6) are well defined.

Let $k = \mathbf{p}_{\text{GHOST}}\langle v \rangle$.

To prove claim (6) we first have to note that the value of ghost_height (and flush_height) is non-decreasing, as is easy to verify by looking at the pseudo-code and specifically at the CheckFlush procedure.

Assume first that k is not the first packet from X that carries ghost information (in addition to ghost packets, donation packets and co-donation packets also carry ghost information). It follows from the Packet Order Axiom and from the monotonicity of ghost_height that $v \geq \text{ghost}[X]_{P@e}$ and we are done.

If k is the first such packet then $\text{ghost}[X]_{P@e}$ is the initial value assigned by the protStart, protRun or the protJoin procedure, where each of the cases occurs when $j(X) = j(P) = 0$; $j(X) \leq j(P) \neq 0$; and $j(X) > j(P)$, respectively.

In case $j(P) = 0$ we have $\text{ghost}[X]_{P@e} = 0 \leq v$ and we are done.

In case $j(X) = j(P) \neq 0$ we have $X = P$ and one can verify by looking at the protRun procedure that

$$\begin{aligned} \text{ghost}[X]_{P@e} &= \text{ghost}[P]_{P@e} = \text{ghost}[P]^{P@v_{j(P)}(P)^{\text{PR}}} = \\ &= \text{ghost_height}^{P@v_{j(P)}(P)^{\text{PR}}} \leq \text{ghost_height}_{P@k^{\text{QU}}} = v \end{aligned}$$

where the last inequality follows from the monotonicity of ghost_height .

Look at the case $j(X) < j(P)$. By induction (on claim (7))

$$\mathit{ghost}[X]_{P@e} = \mathit{ghost}[X]^{P@v_j(P)(P)^{\text{PR}}} \leq \mathit{ghost_height}_{X@v_j(P)(X)^{\text{PR}}}$$

and by the monotonicity of $\mathit{ghost_height}$

$$\mathit{ghost_height}_{X@v_j(P)(X)^{\text{PR}}} \leq \mathit{ghost_height}_{X@k^{\text{QU}}} = v$$

and we are done.

Now look at the case $j(X) > j(P)$. In this case X is a late joining process. Let E be the parent of X . Then

$$\mathit{ghost}[X]_{P@e} = \mathit{ghost}[E]^{P@v_j(X)(P)^{\text{PR}}} \leq \mathit{ghost_height}_{E@v_j(X)(E)^{\text{PR}}}$$

and since X inherits its values of $\mathit{ghost_height}$ from the value of $\mathit{ghost_height}$ in E we have

$$\mathit{ghost_height}_{E@v_j(X)(E)^{\text{PR}}} = \mathit{ghost_height}_{X@v_j(X)(X)^{\text{PR}}}$$

And by monotonicity we have

$$\mathit{ghost_height}_{X@v_j(X)(X)^{\text{PR}}} \leq \mathit{ghost_height}_{X@k^{\text{QU}}} = v$$

and we are done in this case as well.

To prove claim (5) let e' be the trigger of the X -transaction that queued the packet k . Then $e' \prec k^{\text{QU}} \prec k^{\text{PR}} = e$. It follows from Lemma 2 that $\text{view}(e') \leq \text{view}(e)$.

By claim (6) that we just proved and by induction we know that

$$\begin{aligned} \mathit{flush}[X]^{P@e} &= \mathit{flush}[X]_{P@e} \leq \mathit{ghost}[X]_{P@e} \leq \mathit{ghost}[X]^{P@e} = v \leq \mathit{ghost_height}^{X@e'} \leq \\ &\leq (\mathit{cur_view} + \mathit{v_gap})^{X@e'} = \text{view}(e') \leq \text{view}(e) = (\mathit{cur_view} + \mathit{v_gap})^{P@e} \end{aligned}$$

We have to prove the additional assertions in claim (5) in the cases where $\mathit{v_gap}^{P@e} = 0$ and $X \notin \text{ContactSet}^{P@e}$.

In the case $\mathit{v_gap}^{P@e} = 0$ we have by induction

$$\mathit{flush}[X]^{P@e} = \mathit{flush}[X]_{P@e} = (\mathit{cur_view} + \mathit{v_gap})_{P@e} = (\mathit{cur_view} + \mathit{v_gap})^{P@e}$$

The case $X \notin \text{ContactSet}^{P@e}$ cannot occur here. To show that, we only have to prove that $X \notin \mathbb{U}_e$. The rest follows from the fact that $X \in \text{LiveSet}^{P@e}$ and from claim (2).

If $j(X) \geq j(P)$ then $X \notin \mathbb{U}_e$ by definition. If $j(X) < j(P)$ then the first packet that X sends to P is a donation packet which must precede k . Therefore by definition $X \notin \mathbb{U}_e$ and so it must be in ContactSet at this point. This takes care of claim (5).

ReceiveFlush

This procedure updates $\mathit{flush}[X]$ and then calls the `TryToInstall` service procedure. We will start by ignoring `TryToInstall` and show that the inductive hypothesis still holds before `TryToInstall` is invoked. Later we show that `TryToInstall` preserves all the claims.

This procedure affects claims (5), (6) and (8), but for the first two claims only with respect to the sender process X . We know that $\text{LiveSet}_{P@e} = \text{LiveSet}^{P@e}$ and that $X \in \text{LiveSet}^{P@e}$ so both claims must be verified for X .

Also note that it follows from claim (3) that for every process in $\text{LiveSet}^{P@e}$ all the fields in claims (5) and (6) are well defined.

Let $k = \mathbf{p}_{\text{FLUSH}}\langle v \rangle$.

We start with claim (5). To prove it, we first show that $\text{flush}[X]^{P@e} \leq \text{ghost}[X]^{P@e}$. This requires a bit of digging. The packet k is queued by X through the execution of the CheckFlush procedure. This procedure may or may not queue a ghost packet of the same height, to the same target set, immediately prior to queuing the flush packet. If a ghost packet is queued then it follows from the Packet Order Axiom that P processes the ghost packet immediately prior to the current flush packet, resulting in an equality $\text{flush}[X]^{P@e} = \text{ghost}[X]^{P@e}$. The only difficulty arises if a ghost packet is not queued.

The CheckFlush procedure is invoked by X as part of the execution of a notification transaction or an acknowledgement packet processing transaction. An inspection of the pseudo-code easily shows that in the notification case a queuing of a flush packet is always preceded by the queuing of a ghost packet because all three procedures - `protRemove`, `protJoin` and `protRun` - increment $\mathbf{v_gap}$ which results, according to claim (8), in ghost_height being low.

Let e' be the trigger of the X -transaction that queued the packet k . We can assume that e' is an acknowledgement packet processing event. Since e' results in the queuing of a flush packet of height v destined to P without the queuing of a ghost packet of the same height we know by pseudo-code inspection that

$$\begin{aligned} \text{flush_height}_{X@e'} &< (\text{cur_view} + \mathbf{v_gap})_{X@e'} = v \quad \text{and therefore } v > 0 \\ \text{ghost_height}_{X@e'} &= (\text{cur_view} + \mathbf{v_gap})_{X@e'} = v \\ P &\in \text{ContactSet}_{X@e'} \end{aligned}$$

We know from claim (1) that

$$\text{view}(e') = (\text{cur_view} + \mathbf{v_gap})^{X@e'} = (\text{cur_view} + \mathbf{v_gap})_{X@e'} = v$$

Let $f = v_v(X)^{\text{PR}}$ be the most recent view change notification preceding e' . If the notification $v_v(X)$ is a removal or joining of some process $Q \neq X$ then f is not the first event in \mathbb{E}_X and we know by induction from claim (8) that

$$\text{ghost_height}_{X@f} \leq (\text{cur_view} + \mathbf{v_gap})_{X@f} = v - 1 < \text{view}(e')$$

If $f = X_{\text{RUN}}$ then $j(X) = v > 0$ and X has a parent E . The `protRun` procedure does not change the value of ghost_height until CheckFlush is called and therefore by induction

$$\text{ghost_height}_{X@f} = \text{ghost_height}_{E@v_v(E)^{\text{PR}}} \leq (\text{cur_view} + \mathbf{v_gap})_{E@v_v(E)^{\text{PR}}} = v - 1 < \text{view}(e')$$

Taking care to interpret $\text{var}_{X@e'}$ correctly for `protRun` (see Definition 17).

Therefore there is some trigger event $f \preceq f' \prec e'$ at X such that

$$\text{ghost_height}_{X@f'} < \text{ghost_height}^{X@f'} = v$$

Code inspection shows that the transaction $\text{trans}(f')$ must invoke the `CheckFlush` procedure and must result in the queuing of a ghost packet of height v . If P is in the target set of this multicast then we are done. If it is not, then P must join `ContactSet` sometime between $\text{trans}(f')$ and $\text{trans}(e')$. Code inspection shows that P can join `ContactSet` either after a join notification or as a result of sending a donation to X .

By the definition of f there cannot be any notification events between f' and e' and therefore process X must receive a donation packet d from P at some point between these two transactions. This in turn causes X to queue a co-donation packet to P that includes its current `ghost_height` value.

Since $f' \prec d^{\text{PR}} \prec e'$ and since we know by direct code inspection that `ghost_height` is non-decreasing, it follows that the ghost height that X co-donates is equal to v . From the Packet Order Axiom it follows that the co-donation packet is processed by P before k is processed. As a result

$$\text{ghost}[X]_{P@e} = v$$

And we are done showing that $\text{flush}[X]^{P@e} \leq \text{ghost}[X]^{P@e}$. By induction we can conclude (using claim (5)) that

$$\text{ghost}[X]^{P@e} = \text{ghost}[X]_{P@e} \leq (\text{cur_view} + \text{v_gap})_{P@e} = (\text{cur_view} + \text{v_gap})^{P@e}$$

Which concludes the proof of the first part of claim (5).

We have to prove the additional assertions in claim (5) in the cases where $\text{v_gap}^{P@e} = 0$ and $X \notin \text{ContactSet}^{P@e}$. The case $X \notin \text{ContactSet}^{P@e}$ cannot occur here for the same reason as in the case of `ReceiveGhost`.

If $\text{v_gap} = 0$ then we can assume by induction that

$$\text{flush}[X]_{P@e} = (\text{cur_view} + \text{v_gap})_{P@e}$$

We now use claim (6) (which we will prove shortly) to conclude that

$$\begin{aligned} (\text{cur_view} + \text{v_gap})^{P@e} &= (\text{cur_view} + \text{v_gap})_{P@e} = \text{flush}[X]_{P@e} \leq \\ &\leq \text{flush}[X]^{P@e} \leq (\text{cur_view} + \text{v_gap})^{P@e} \end{aligned}$$

And we are done.

The proof that the `ReceiveFlush` procedure preserves (6) is almost identical to the same proof for the `ReceiveGhost` procedure. The only difference is that in the case where k is the first packet that carries flush information and where $j(X) > j(P)$, the process X inherits its value of `flush_height` from the value of `ghost_height` at the parent E , and not the value of `flush_height` of the parent. Similarly process P initializes `flush[X]` from `ghost[E]` and not from `flush[E]`. Therefore we get

$$\text{flush}[X]_{P@e} = \text{ghost}[E]^{P@v_j(X)(P)^{\text{PR}}} \leq \text{ghost_height}_{E@v_j(X)(E)^{\text{PR}}}$$

and since X inherits its values of *flush_height* from the value of *ghost_height* in E we have

$$\text{ghost_height}_{E@v_j(X)(E)^{\text{PR}}} = \text{flush_height}_{X@v_j(X)(X)^{\text{PR}}}$$

And by monotonicity we have

$$\text{flush_height}_{X@v_j(X)(X)^{\text{PR}}} \leq \text{flush_height}_{X@k^{\text{QU}}} = v$$

and we are done.

Proof that the procedure preserves (8) is only required when $X = P$, i.e. when k is a self packet. In that case we can use the monotonicity of *flush_height* to conclude

$$\text{flush}[P]^{P@e} = v = \text{flush_height}_{P@k^{\text{QU}}} \leq \text{flush_height}^{P@k^{\text{PR}}} = \text{flush_height}^{P@e}$$

Finally the procedure invokes the TryToInstall procedure which preserves all the claims as we show further on.

protStart

It is trivial to check that the procedure initializes P to a state that conforms to the requirements of the lemma.

protRemove

Let R be the process that is being removed. Then by definition $r(R) = \text{view}(e)$.

To see that this procedure preserves (1) notice that its event e is a notification event, meaning that if e' is the preceding event in \mathbb{E}_P then $\text{view}(e) = \text{view}(e') + 1$ while the procedure increments v_gap , maintaining the equality.

It preserves the first part of claim (2) because it removes R from LiveSet. R is the only process in LiveSet that no longer meets the condition $j(Q) \leq \text{cur_view} + v_gap < r(Q)$ as v_gap is incremented. No other process is affected because no other process satisfies either $r(Q) = r(R)$ or $j(Q) = r(R)$.

It preserves the second part of claim (2) because it removes R from LiveSet and ContactSet, while not affecting \mathbb{U}_e since e is not a donation packet dequeuing event.

It preserves (3) because it removes the R coordinate from all the required vectors. It does not affect (4).

It preserves (5) because it shrinks LiveSet, makes $v_gap > 0$ and increments the right hand side of the inequality while not affecting the left hand side.

To see why it preserves claims (8) and (9), notice that by incrementing v_gap it forces *ghost_height* and *flush_height* to be low without touching them. As a result the call to CheckFlush at the end has the following effects:

- *ghost_height* becomes high if and only if $\text{FwdWaitSet} = \emptyset$.
- *flush_height* becomes high if and only if $\text{WaitSet} = \emptyset$.

Therefore *flush_height* becomes high only if *ghost_height* becomes high, and therefore by induction $\text{flush_height} \leq \text{ghost_height}$ in all cases.

The procedure increments *v_gap*, making (10) vacuously true.

It preserves (6) because it does not change the values of *ghost*[*X*] and *flush*[*X*] for $X \neq R$.

Finally we have to show that the procedure preserves claim (7). Let $i = \text{view}(e)$ and let $X \neq R$ be any process that remains live after the view change. If P does not dequeue any ghost (flush), donation or co-donation packet from X between $v_{i-1}(P)^{\text{PR}}$ and $e = v_i(P)^{\text{PR}}$ then we get by induction, for either ghost or flush

$$\begin{aligned} \text{ghost}[X]^{P@v_i(P)^{\text{PR}}} &= \text{ghost}[X]^{P@v_{i-1}(P)^{\text{PR}}} \leq \text{ghost_height}_{X@v_{i-1}(X)^{\text{PR}}} \leq \text{ghost_height}_{X@v_i(X)^{\text{PR}}} \\ \text{flush}[X]^{P@v_i(P)^{\text{PR}}} &= \text{flush}[X]^{P@v_{i-1}(P)^{\text{PR}}} \leq \text{flush_height}_{X@v_{i-1}(X)^{\text{PR}}} \leq \text{flush_height}_{X@v_i(X)^{\text{PR}}} \end{aligned}$$

and we are done. Otherwise, let $e' = k^{\text{PR}}$ be the last dequeuing event of a ghost (flush), donation or co-donation packet from X at P with $\text{view}(e') = i - 1$. Let v be the ghost (flush) height carried by the packet. Then we have in each case respectively

$$\begin{aligned} \text{ghost}[X]^{P@v_i(P)^{\text{PR}}} &= \text{ghost}[X]^{P@e'} = v = \text{ghost_height}_{X@k^{\text{QU}}} \\ \text{flush}[X]^{P@v_i(P)^{\text{PR}}} &= \text{flush}[X]^{P@e'} = v = \text{flush_height}_{X@k^{\text{QU}}} \end{aligned}$$

and from the Piggyback Axiom it follows that $k^{\text{QU}} \prec v_i(X)^{\text{PR}}$ and so in each case respectively

$$\begin{aligned} \text{ghost_height}_{X@k^{\text{QU}}} &\leq \text{ghost_height}_{X@v_i(X)^{\text{PR}}} \\ \text{flush_height}_{X@k^{\text{QU}}} &\leq \text{flush_height}_{X@v_i(X)^{\text{PR}}} \end{aligned}$$

and we are done.

protJoin

Let J be the joining process and let E be its parent. Then by definition $j(J) = \text{view}(e)$.

This procedure preserves (1), (8), (9) and (10) for the exact same reasons as *protRemove*.

it satisfies the first part of claim (2) because it adds J to *LiveSet*. J is the only process that newly meets the condition $j(Q) \leq \text{cur_view} + \text{v_gap} < r(Q)$ as *v_gap* is incremented. No other process is affected, because no other process satisfies either $j(Q) = j(J)$ or $r(Q) = j(J)$.

It preserves the second part of claim (2) because it adds J to *LiveSet* and *ContactSet*, while not affecting \mathbb{U}_e since e is not a donation packet dequeuing event.

It preserves (3) because it adds a J coordinate to all the required vectors. It does not affect (4).

It preserves (5) because it makes *v_gap* > 0 and increments the right hand side of each existing inequality while not affecting the left hand side. The new inequalities for the J coordinate are inherited from the original ghost inequality for its parent E .

It preserves (6) because it does not change the values of *ghost*[*X*] and *flush*[*X*] for $X \neq J$.

Finally we have to show that the procedure preserves claim (7). Let $i = \text{view}(e)$ and let X be any process that is live after the view change. If $X \neq J$ then the claim holds following the exact same argument that we used in the `protRemove` case. In the case $X = J$ we have

$$\text{flush}[J]^{P@v_i(P)^{\text{PR}}} = \text{ghost}[J]^{P@v_i(P)^{\text{PR}}} = \text{ghost}[E]^{P@v_i(P)^{\text{PR}}}$$

and since we have already proven the case $X \neq J$ we know that

$$\text{ghost}[E]^{P@v_i(P)^{\text{PR}}} \leq \text{ghost_height}_{E@v_i(E)^{\text{PR}}}$$

Since J inherits its values `ghost_height` and `flush_height` from its parent value of `ghost_height` we have

$$\text{ghost_height}_{E@v_i(E)^{\text{PR}}} = \text{ghost_height}_{J@v_i(J)^{\text{PR}}} = \text{flush_height}_{J@v_i(J)^{\text{PR}}}$$

and we are done.

Notice that in the last equation by definition the value $\text{flush_height}_{J@v_i(J)^{\text{PR}}}$ reflects the fact that the `protRun` procedure copies the initial value of `ghost_height` into `flush_height` (see Definition 17).

protRun

Let J be the new process and let E be its parent. Then by definition $j(J) = \text{view}(e)$. Let $e_E = v_{j(J)}(E)^{\text{PR}}$ and let $e' \prec e_E$ be the event immediately preceding e_E in E . It follows from the Parent Axiom that e_E exists and therefore e' exists as well since e_E is not the first event in \mathbb{E}_E .

Process J starts life with the exact same state that its parent had when it dequeued the `n_JOIN(J, E)` notification, which is the same state it had at the conclusion of the `trans(e')`. By induction, E satisfied all the claims at that point. For most claims this means that they are automatically satisfied at that point in J as well, as long as we interpret the expressions $\text{var}^{J@e'}$ to simply mean the initial value of `var`, ignoring the fact that the value of e' there is undefined in J . However there are a number of exceptions.

claim (1) becomes ill-defined because e' is not defined in J . The second part of claim (2) is not satisfied because it depends on the value of $\mathbb{U}_{e'}$ which is ill-defined at J , and any rate is not inherited from E . Claim (6) becomes ill-defined because `LiveSet` $_{J@e'}$ has no clear interpretation in J . Similarly, claim (7) depends on the meaning of e' and so is hard to interpret in J . Claim (8) references the self value `flush[P]` where P is the local process. Whenever we need rely on any of these ill-defined inductive claims as we go, we will present an explicit calculation that relies directly on the well-defined E version of these claims.

The `protRun` procedure preserves (1) because

$$\begin{aligned} \text{view}(e) &= j(J) = \text{view}(e') + 1 = (\text{cur_view} + \text{v_gap})^{E@e'} + 1 = \\ &= (\text{cur_view} + \text{v_gap})^{J@e'} + 1 = (\text{cur_view} + \text{v_gap})^{J@e} \end{aligned}$$

Where the last equality follows because the `protRun` procedure increments `v_gap`.

claims (9), (10) and most of (8) are preserved by `protRun` for the exact same reasons as in the `protRemove` case. The only missing piece is that the inequality

$$\text{flush}[J]^{J@e} \leq \text{flush_height}^{J@e}$$

follows because the self flush height $\text{flush}[J]$ is initialized to be equal to $\text{ghost_height}^{J@e'}$. The same value is used to initialize flush_height , and thereafter flush_height can only increase.

`protRun` satisfies the first part of claim (2) for the same reason that `protJoin` does.

`protRun` satisfies the second part of claim (2) because at the outset \mathbb{U}_e contains every process X whose join view is lower than $j(J)$. This includes every member of `LiveSet` except J itself. Since the `protRun` procedure adds J to `LiveSet` while setting `ContactSet` = $\{J\}$, this makes the equations true.

It preserves (3) because it adds a J coordinate to all the required vectors. It does not affect (4)

It preserves (5) because it makes $v_gap > 0$ and increments the right hand side of each existing inequality while not affecting the left hand side. The new inequalities for the J coordinate hold because the values of $\text{flush}[J]$ and $\text{ghost}[J]$ are both initialized to $\text{ghost_height}^{J@e'} \leq \text{ghost_height}^{J@e}$ and because we have shown that claim (8) is preserved.

It preserves (6) because the `protRun` procedure does not change the initial values of the $\text{ghost}[]$ and $\text{flush}[]$ vectors.

Finally we have to show that the procedure preserves claim (7). Let $i = \text{view}(e)$ and let X be any process that is live after the view change. If $X \neq J$ then the value of $\text{ghost}[X]$ and $\text{flush}[X]$ is inherited from E without change

$$\begin{aligned} \text{ghost}[X]^{J@v_i(J)^{\text{PR}}} &= \text{ghost}[X]_{E@v_i(E)^{\text{PR}}} \\ \text{flush}[X]^{J@v_i(J)^{\text{PR}}} &= \text{flush}[X]_{E@v_i(E)^{\text{PR}}} \end{aligned}$$

and since we have already proved all the claims for E (the `protJoin` case) we know that

$$\begin{aligned} \text{ghost}[X]_{E@v_i(E)^{\text{PR}}} &\leq \text{ghost}[X]^{E@v_i(E)^{\text{PR}}} \leq \text{ghost_height}_{X@v_i(X)^{\text{PR}}} \\ \text{flush}[X]_{E@v_i(E)^{\text{PR}}} &\leq \text{flush}[X]^{E@v_i(E)^{\text{PR}}} \leq \text{flush_height}_{X@v_i(X)^{\text{PR}}} \end{aligned}$$

and we are done. The case $X = J$ follows because the `protRun` procedure sets

$$\text{ghost}[J]^{J@v_i(J)^{\text{PR}}} = \text{flush}[J]^{J@v_i(J)^{\text{PR}}} = \text{ghost_height}_{J@v_i(J)^{\text{PR}}} = \text{flush_height}_{J@v_i(J)^{\text{PR}}}$$

ReceiveDonation

Let S be the sender of the donation packet. Since e is a donation packet dequeuing event in this case, S falls out of \mathbb{U}_e . This is compensated for by adding S to `ContactSet`. This preserves claim (2). Then the procedure makes a sequence of invocations of the `ReceiveMessage` and `ReceiveAck` procedures which preserve all the claims as we have already shown.

Finally the procedure updates $ghost[S]$ and $flush[S]$ from the donated values of $ghost_height$ and $flush_height$ respectively. Remember that the donation packet is sent by the $protJoin$ procedure as part of the $j(P)$ view change notification processing. Therefore

$$\begin{aligned} ghost[S]^{P@e} &= ghost_height_{S@v_{j(P)}(S)^{PR}} \\ flush[S]^{P@e} &= flush_height_{S@v_{j(P)}(S)^{PR}} \end{aligned}$$

It follows from claim 8 of the inductive hypothesis that

$$\begin{aligned} flush_height_{S@v_{j(P)}(S)^{PR}} &\leq ghost_height_{S@v_{j(P)}(S)^{PR}} \leq (cur_view + v_gap)_{S@v_{j(P)}(S)^{PR}} = \\ &= j(P) - 1 < view(e) = (cur_view + v_gap)^{P@e} \end{aligned}$$

This proves most of claim 5. Since the inequalities are strict we have to show that $v_gap > 0$. It follows from claim (6) which we will prove shortly that $flush[S]$ is non-decreasing, and therefore the inequality was strict at the start of the transaction. Therefore by induction $v_gap > 0$.

Claim (6) follows by the exact same reasoning that we used for the $ReceiveGhost$ and $ReceiveFlush$ procedures, while claim (7) is not relevant at a non-notification transaction.

ReceiveCoDonation

Let S be the sender of the co-donation packet, and let e' be the trigger of the donation transaction at which S queues the co-donation packet.

The $ReceiveCoDonation$ procedure makes a sequence of calls that preserve all the claims and then updates $ghost[S]$ and $flush[S]$ from the co-donated values. Then

$$\begin{aligned} ghost[S]^{P@e} &= ghost_height_{S@e'} \\ flush[S]^{P@e} &= flush_height_{S@e'} \end{aligned}$$

From claim 8 of the inductive hypothesis and the Piggyback Axiom we know that

$$\begin{aligned} flush_height_{S@e'} &\leq ghost_height_{S@e'} \leq (cur_view + v_gap)_{S@e'} \leq \\ &\leq view(e') \leq view(e) = (cur_view + v_gap)^{P@e} \end{aligned}$$

which proves most of claim (5). Process S must be contacted at this point, so we do not have to show that the inequalities are strict. But if one of the inequalities happens to be strict, then we have to show that $v_gap > 0$. This follows from the same argument as in the case of $ReceiveDonation$.

Claim (6) follows by the exact same reasoning that we used for the $ReceiveGhost$ and $ReceiveFlush$ procedures, while claim (7) is not relevant at a non-notification transaction.

Finally the procedure calls $TryToInstall$ which also preserves all the claims as we will see.

TryToInstall

This is a service procedure that is called by other procedures. However it preserves (1) and (5) because it always increments *cur_view* and decrements *v_gap* together. Also, the procedure makes no changes to *v_gap* unless all the *flush[]* values are high. It preserves (4) because it resets *vt* whenever it changes *MSet*.

It preserves (8) because it does not touch either *ghost_height*, *flush_height* or the self flush height *flush[P]* while keeping *cur_view* + *v_gap* fixed (this includes possible invocations of the *protBroadcast* and *Scan* procedures).

It preserves (9) because it either results in *v_gap* = 0 which makes (9) vacuously true, or else it fails to install views in which case it does not change any variables and therefore preserves (9) by induction.

It preserves (10) because it either results in *v_gap* > 0 which makes (10) vacuously true, or else it starts out with *v_gap* = 0 in which case (10) is true by induction, or else it starts out with *v_gap* > 0 and ends with *v_gap* = 0 in which case it empties out *LaunchQueue*, making (10) true.

TryToInstall does not affect claims (2) and (3).

Scan

This is a service procedure that is called by other procedures. It does not affect any of the relevant variables so it has no effect on any of the claims.

□

Corollary 5. *Let P be a process and let $e \in \mathbb{E}_P$ be any trigger event. Suppose that*

$$\text{LiveSet}_{P@e} \neq \text{ContactSet}_{P@e}$$

Then there is a process $X \in \text{LiveSet}$ such that $\text{flush}[X] < j(P)$.

Proof. It follows from Lemma 8(2) that there is a live process $X \in \text{LiveSet} \cap \mathbb{U}_e$, and it follows from the definition of \mathbb{U}_e that $j(X) < j(P)$. Therefore P is not an original process and it has a parent E . P starts life with a state identical to the state of E at $v_{j(P)}(E)^{\text{PR}}$.

Process X can only be in *LiveSet* if it is there originally or if P receives a join notification for X . Since the latter does not happen in this case we must have $X \in \text{LiveSet}_{E@v_{j(P)}(E)^{\text{PR}}}$ and by Lemma 8(3) $\text{flush}[X]_{E@v_{j(P)}(E)^{\text{PR}}}$ exists. By Lemma 8(6, 7, 8 and 1)

$$\begin{aligned} \text{flush}[X]_{E@v_{j(P)}(E)^{\text{PR}}} &\leq \text{flush}[X]^{E@v_{j(P)}(E)^{\text{PR}}} \leq \\ &\leq \text{flush_height}_{X@v_{j(P)}(X)^{\text{PR}}} \leq (\text{cur_view} + \text{v_gap})_{X@v_{j(P)}(X)^{\text{PR}}} < j(P) \end{aligned}$$

Therefore the initial value of *flush[X]* at P is smaller than $j(P)$.

Scanning the pseudo-code reveals that this initial value of *flush[X]* does not change in P unless P receives a notification of the removal of X (which does not happen in this case) or if P processes a co-donation packet from X (which also does not happen, because co-donations are always sent from a late joining process to an existing group member), or if P processes a flush or a donation

packet from X . The donation packet to P is the first packet that X queues to \overrightarrow{XP} and therefore the first packet from X that P processes. Since e occurs before the donation packet is processed, the initial value of $\text{flush}[X]$ is still extant and we are done. \square

Lemma 9. *Let P and Q be two original processes. Suppose that P sends a message packet $\mathbf{p}_{\text{MSG}}\langle \text{msg} \rangle$ to Q . When P queues the packet, it increments its mpkt_out and places msg in its WaitSet together with an index value equal to its updated value of mpkt_out (see the protBroadcast procedure for original message broadcasts and the protRemove procedure for forwarded broadcasts). If Q processes the message it increments the value of $\text{mpkt_in}[P]$. Then:*

1. *Before Q processes the message, its value of $\text{mpkt_in}[P]$ is lower than index .*
2. *After Q processes the message, its value of $\text{mpkt_in}[P]$ becomes equal to index .*

Moreover, if $P = Q$ then the conclusion holds without the requirement that P be original.

Proof. Both P and Q are original. Therefore Q is a member of $\text{ContactSet}(P)$ from the start (see the protStart procedure). Therefore every message that was queued by P prior to msg had Q in its recipient list and since channels are FIFO, all of these messages are processed by Q before msg is processed. The mpkt_out variable in P is incremented every time a message packet is multicast by P (at the $.b$ or $.f$ component, according as the message is original or forwarded), and similarly $\text{mpkt_in}[P]$ is incremented by Q every time a message from P is processed (at the $.b$ or $.f$ component, according as the message is original or forwarded). Initially both variables are equal to zero (at both components) at both P and Q (see the protStart procedure). Therefore they remain at lockstep as claimed.

Almost the same argument works when $P = Q$ and P is not an original process. We just have to verify two things. One, that $P \in \text{ContactSet}(P)$ from the start, as one can verify by looking at the protRun procedure. Two, that initially both mpkt_out and $\text{mpkt_in}[P]$ are equal at P . This can also be verified by looking at the protRun procedure. \square

Lemma 10. *Let P and Q be two original processes. Suppose that P sends a flush packet $k = \mathbf{p}_{\text{FLUSH}}\langle v \rangle$ to Q . When P queues the packet, it increments its flush_height (see the CheckFlush procedure). If Q processes the flush it increments the value of $\text{flush}[P]$. Then:*

$$\text{flush}[P]_{Q@k^{\text{PR}}} < \text{flush_height}_{P@k^{\text{QU}}} = \text{flush}[P]^{Q@k^{\text{PR}}} = v$$

Moreover, if $P = Q$ then the conclusion holds without the requirement that P be original.

The exact same claim is true if flush is replaced by ghost throughout.

Proof. We prove the lemma for the flush case. The ghost case is identical using the appropriate substitutions. Both P and Q are original. Therefore Q is a member of $\text{ContactSet}(P)$ from the start (see the protStart procedure). Therefore every flush that is queued by P prior to $\mathbf{p}_{\text{FLUSH}}\langle v \rangle$ had Q in its recipient list and since channels are FIFO, all of these flushes are processed by Q before $\mathbf{p}_{\text{FLUSH}}\langle v \rangle$ is processed. The flush_height variable in P is increased to be equal to w every time a flush of height w is broadcast by P (see the CheckFlush procedure), and therefore $\text{flush}[P]$ is increased by Q every time a flush from P is processed (see the ReceiveFlush procedure). Initially both variables are equal to zero at both P and Q (see the protStart procedure). Therefore they remain at lockstep as claimed.

Almost the same argument works when $P = Q$ and P is not an original process. We just have to verify two things. One, that $P \in \text{ContactSet}(P)$ from the start, as one can verify by looking at the `protRun` procedure. Two, that initially both flush_height and $\text{flush}[P]$ are equal at P . This can also be verified by looking at the `protRun` procedure. \square

4.2 Side Effects of CBCAST Triggers

As we have seen, each trigger event - a notification event, a packet processing event, or a message broadcast request event - causes a CBCAST callback to be invoked. Each invocation can cause zero or more packets to be queued on various channels - in other words the invocation causes side effects (see 2.2). We are now going to characterize the side effects of each type of trigger in detail.

4.2.1 Side effects of message broadcast request events

Message broadcast requests are processed through the `protBroadcast` procedure. The following lemma details the possible side effects of this procedure call.

Lemma 11. *An invocation of the `protBroadcast` procedure results in exactly one of the following outcomes:*

- *No additional packet queuing, if $v_gap > 0$.*
- *A message packet multicast if $v_gap = 0$.*

Proof. Obvious from the code of `protBroadcast`. \square

4.2.2 Side effects of view change notifications

A view change notification from GMS is processed either through the `protJoin` or the `protRemove` procedures, depending on the type of view change. In addition, a joining process starts out life with an exact replica of the state of its parent, and immediately invokes the `protRun` procedure. The following lemma details the possible effects of these calls.

Lemma 12. *1. An invocation of the `protJoin` procedure results in the queuing of a donation packet, followed by exactly one of the following outcomes:*

- *No additional packet queuing, if `FwdWaitSet` is not empty.*
- *A ghost packet multicast if `FwdWaitSet` is empty and `BcastWaitSet` is not empty.*
- *A ghost packet multicast followed by a flush packet multicast, if both `BcastWaitSet` and `FwdWaitSet` are empty.*

2. An invocation of the `protRemove` procedure when process P is removed, results in exactly one of the following outcomes:

- *A sequence of message broadcasts, one per message in `FwdQueue[P]`, if `FwdQueue[P]` is not empty.*

- No additional packet queuing if $FwdQueue[P]$ is empty, and $FwdWaitSet$ is non-empty when $CheckFlush$ is called.
 - A ghost packet multicast if $FwdWaitSet$ is empty and $BcastWaitSet$ is not empty when $CheckFlush$ is called.
 - A ghost packet multicast followed by a flush packet multicast if both $BcastWaitSet$ and $FwdWaitSet$ are empty when $CheckFlush$ is called.
3. An invocation of the $protRun$ procedure results in exactly one of the following outcomes:
- No additional packet queuing if $FwdWaitSet$ is non-empty.
 - A ghost packet multicast followed by a flush packet multicast if $FwdWaitSet$ is empty.

Proof. All these outcomes are easy to verify by tracing the code path in the respective calls. In the case of $protRemove$ it is important to notice that if $FwdQueue[P]$ is not empty, then the forwarded messages are placed in $FwdWaitSet$, which as a result is not empty when $CheckFlush$ is called. \square

4.2.3 Side effects of message and acknowledgement packet receipts

A message receipt always results in the sending of an acknowledgement packet in response. An acknowledgement packet receipt causes a stabilization of a message with respect to the process that sent the packet. If this stabilization causes either $FwdWaitSet$ or $BcastWaitSet$ to empty out, it can cause the multicasting of ghost and flush packets.

Lemma 13. 1. An invocation of the $ReceiveMessage$ procedure results in the queuing of an acknowledgement packet targeted at the sender of the message.

2. An invocation of the $ReceiveAck$ procedure results in exactly one of the following outcomes:
- No additional packet queuing if $v_gap = 0$.
 - No additional packet queuing if the acknowledgement does not cause either $FwdWaitSet$ or $BcastWaitSet$ to empty out, even if one or both were already empty.
 - No additional packet queuing if the acknowledgement causes $BcastWaitSet$ to empty out and $FwdWaitSet$ is non-empty.
 - A ghost packet multicast if $v_gap > 0$ and the acknowledgement causes $FwdWaitSet$ to empty out and $BcastWaitSet$ is non-empty.
 - A flush packet multicast if $v_gap > 0$ and the acknowledgement causes $BcastWaitSet$ to empty out and $FwdWaitSet$ is empty.
 - A ghost packet multicast followed by a flush packet multicast, if $v_gap > 0$ and the acknowledgement causes $FwdWaitSet$ to empty out and $BcastWaitSet$ is empty.

Proof. This follows directly from direct observation and from Lemma 8(5 and 9). \square

4.2.4 Side effects of ghost and flush packet receipts

A ghost packet receipt does not cause any other packets to be sent. A flush packet, however, may cause one or more views to be installed. If that happens then one or more original messages from `LaunchQueue` may be broadcast.

Lemma 14. 1. *An invocation of the `ReceiveGhost` procedure does not result in additional packet queuing.*

2. *An invocation of the `ReceiveFlush` procedure results in exactly one of the following outcomes:*

- *No additional packet queuing if no views are installed or if `LaunchQueue` is empty.*
- *One or more message packet multicasts if one or more pending views are installed and `LaunchQueue` is not empty.*

Proof. This follows from Lemma 8(10) and from direct inspection of the code of the relevant procedures. \square

4.2.5 Side effects of donation and co-donation packet receipts

The side effects of invoking the `ReceiveDonation` procedure are pretty straightforward - first a co-donation packet is sent and then a sequence of `ReceiveMessage` and `ReceiveAck` invocations are performed, each with its side effects that have already been characterized. The side effects of invoking the `ReceiveCoDonation` procedure are a bit more subtle, because this procedure has an additional call to `TryToInstall` at the end and there is an interplay between its side effects and the side effects of the rest of the procedure.

Lemma 15. 1. *An invocation of the `ReceiveDonation` procedure results in the queuing of a co-donation packet back to the sender of the donation packet, followed by a sequence of side effects for each of the `ReceiveMessage` and `ReceiveAck` invocations.*

2. *An invocation of the `ReceiveCoDonation` procedure results in exactly one of the following outcomes:*

- *Zero or more invocations of `ReceiveMessage` or `ReceiveAck` occur with their side effects, and `TryToInstall` fails to install a new view and has no side effects.*
- *No invocations of either `ReceiveMessage` or `ReceiveAck` occur, and `TryToInstall` succeeds in installing all the views. As a result zero or more original messages from `LaunchQueue` are broadcast.*

Proof. The donation case is self evident. In the co-donation case, if $v_gap = 0$ (we will see later that this case does not actually happen) then `TryToInstall` does not install any views and it follows from Lemma 8(10) that `LaunchQueue` is empty and so `TryToInstall` has no side effects.

Therefore the only non-trivial case has to do with a co-donation that starts with $v_gap > 0$ and results in a successful new view installation. Suppose that the sender of the co-donation is G and

the receiver is P . Then a successful installation requires, at P :

$$\begin{aligned} flush[G] &= cur_view + v_gap \\ flush[P] &= cur_view + v_gap \end{aligned}$$

From Lemma 8(8) we know that at P

$$flush[P] \leq flush_height \leq cur_view + v_gap$$

Therefore $flush[P] = flush_height = cur_view + v_gap$ and since $v_gap > 0$ it follows from the same lemma (part 9) that $WaitSet$ is empty and as a result UNT_P is empty.

Looking at the `ReceiveCoDonation` procedure one sees that P updates its value of $flush[G]$ just before invoking `TryToInstall`, setting it to be equal to the co-donated value of $flush_height$ in G . Therefore, at the moment that G sends the co-donation, it has $flush_height(G) = cur_view(P) + v_gap(P)$. In other words, if we denote by d the donation packet that P sends to G and by c the co-donation packet that G sends to P then

$$flush_height_{G@dPR} = (cur_view + v_gap)_{P@cPR}$$

The Piggyback Axiom and Lemma 8(1) imply that the co-donation packet cannot be processed when P has an overall view height that is lower than that of G and therefore

$$(cur_view + v_gap)_{P@cPR} \geq (cur_view + v_gap)_{G@dPR}$$

therefore

$$flush_height_{G@dPR} \geq (cur_view + v_gap)_{G@dPR}$$

From Lemma 8(8) it follows that

$$flush_height_{G@dPR} \leq (cur_view + v_gap)_{G@dPR}$$

and so we can conclude that

$$flush_height_{G@dPR} = (cur_view + v_gap)_{G@dPR}$$

From Lemma 8(2 and 1) it follows that $P \notin ContactSet_{G@dPR}$. From Corollary 5 it follows that $flush[P]_{G@dPR} < (cur_view + v_gap)_{G@dPR}$. From Lemma 8(5) it follows that $v_gap_{G@dPR} > 0$.

Now we can use Lemma 8(9) to conclude that $WaitSet_{G@dPR}$ is empty. Therefore P receives an empty $WaitSet$ from G as part of its co-donation, and therefore UNT_g is empty and we are done. \square

4.3 CBCAST is Vacuum Convergent

In our analysis of CBCAST we want to take advantage of the main finding of Section 2, namely the Fault Theorem (Theorem 1), and limit our attention to transactional histories. In order to do that we have to prove that CBCAST is vacuum convergent (see Definitions 15).

Theorem 3. *The CBCAST protocol is vacuum convergent.*

Proof. Let P be a halting process in a **CBCAST** based conforming history. Look at step (6) of the vacuum loop (see Definition 12). By the Conforming **GMS** Axiom process P has a finite view interval and therefore this step can only increment v a finite number of times. Afterwards either the loop exits or step (6) is not executed again. Assume that the loop never exits. Once v stabilizes, steps (3) and (4) are executed once and are not executed again. Therefore after a finite number of steps the vacuum loop degenerates to repeated executions of step (5) which consist of: processing packets on the self channel; queuing side-effect packets to their respective channels, including the self-channel; sending and receiving the queued packets on downstream channels, including the self-channel; processing the newly received packets on the self-channel; etc. Since donation and co-donation packets are never queued to the self channel, at this point such packets are not processed by the vacuum loop anymore.

From Lemma 8 we know that the values of *ghost_height* and *flush_height* in P cannot rise above v . Since the CheckFlush procedure only generates ghost and flush packets when the ghost and flush height rise, the vacuum loop can only generate a finite number of such packets. Therefore after a finite number of steps no such packets are generated anymore. Therefore after some more time passes the vacuum loop processes the last of these packets and does not process any ghost or flush packets afterwards.

Message packets are generated as a result of the processing of

- a message broadcast request (when $v_gap = 0$)
- a flush packet (when the flush causes view installations and **LaunchQueue** is not empty)
- a co-donation packet (ditto)
- a removal notification (when **FwdQueue** is not empty)

Since at this point the vacuum loop does not process any additional items of these types, it only accumulates a finite number of message packets and as a result after some point it processes the last message packet and does not process any more such packets afterwards.

Acknowledgement packets are generated as a result of the processing of message packets, donation packets and co-donation packets. Therefore the vacuum loop processes a finite number of those packets as well.

So at some point the vacuum loop runs out of packets to process and is forced to fall through to step (6), contrary to our assumption. \square

5 The History Reduction Mapping

5.1 Introduction

In this section we demonstrate the rather surprising fact that any transactional history of the **CBCAST** protocol that contains at least one join notification can be reduced to an alternate conforming history of **CBCAST** that performs the same computation and where the first join notification is replaced with a removal notification. This allows us to restrict our analysis to histories that contain no join notifications.

We will construct the reduced history explicitly. We will start with the original history, and make careful changes to it. The construction will revolve around the first joining process G , its parent D , and the join view of G , which we call the *critical view*. The idea is to declare that G is an original group member (member since view zero) and that it has a doppelgänger $-G$ that is also a member of view zero. Then instead of having G join the group, we have $-G$ leave the group. We need to accomplish this while not violating the CBCAST protocol, and without affecting the user application. In fact if we have any hope of success, the user application must be completely oblivious to the change. To create the reduced history we will have to solve two problems. First we will have to create a whole new history for $\pm G$ during the pre-critical interval, namely the interval that precedes the critical view change. Then we will have to resolve the race conditions that occur as a result of *untimely* packets, namely packets that are sent before the critical view change notification and arrive afterwards.

We will solve the first problem by using D as the pre-critical template for $\pm G$. This means that every reactive move by D , like receiving or acknowledging a message, will be copied by $\pm G$ verbatim. However we will not copy proactive moves by D , namely original message broadcasts that are initiated by the application at D . During the pre-critical period, $\pm G$ will be passive participants. We will ensure that the APP thread does not run there (and therefore does not generate message queuing requests) by artificially delaying the launch of the APP thread until after the critical moment. This is possible to arrange because the launch of the APP thread in `protStart` is asynchronous.

The second problem will be solved with the help of the donation/co-donation protocol. This protocol is carefully tailored to provide precise compensation for critical boundary race conditions. In the reduced history the critical donation and co-donation packets will be eliminated. All the simulated packet processing that occurs during the `ReceiveDonation` and `ReceiveCoDonation` procedure calls will be replaced with the receipt and processing of actual, newly-minted untimely packets.

The new packets and events will have to be added and timed in a very precise manner so that we neither violate CBCAST nor affect the user application. For example, pre-critical flush packets from $\pm G$ will have to arrive slightly earlier than their counterparts from D , in order to prevent them from causing the receiving process to install a new view. But pre-critical forwarded messages from $\pm G$ will have to arrive slightly *later* than their counterparts from D , in order to guarantee that they are redundant, and therefore ignored by the receiving process. We do not want $\pm G$ to rock the boat prematurely.

To manage this careful surgery we need quite a bit of control. We will gain this control by creating a 4-coordinate label for each event. The label will describe which transaction the event belongs to and whether it is the transaction trigger or a side effect. It will describe whether the event was moved slightly forward or backward to insure redundancy. For events that occur during a donation or co-donation transaction, the label will also describe which simulated sub-transaction the event is related to. The most important fact about the labels is that they all come from a single partially ordered *label space* that is common to the original and reduced histories. The use of a single label space for both histories will allow us to relate the order in both, and ultimately to use induction over the shared label space to prove that both histories track each other closely and ultimately converge.

Throughout this part of the paper H is a fixed transactional history that includes a first join view v_{crit} . The process that joins at view v_{crit} is denoted G throughout, and the parent process of G is denoted by D . We will sometimes refer to H as the *original history*. We will sometimes refer to

the synthesized history H^r as the *reduced history*.

5.2 Preliminaries

5.2.1 Transactions

A process executing the **CBCAST** protocol invokes a procedure in reaction to each triggering event. A view change notification event triggers an invocation of the `protStart`, `protRun`, `protJoin` or `protRemove` procedures. A packet processing event causes the invocation of the `ReceiveMessage`, `ReceiveAck`, `ReceiveGhost`, `ReceiveFlush`, `ReceiveDonation` or `ReceiveCoDonation` procedures. A message broadcast request event causes an invocation of the `protBroadcast` procedure. The procedure call in turn causes zero or more side effects in the form of packet queuing events. This sequence of events, starting with a trigger and continuing with a finite number (possibly zero) of side effects is a *transaction*. Because each process P runs its **CBCAST** procedures in a critical section, the events at each transaction occur as a contiguous sequence in \mathbb{E}_P and therefore transactions can be read out of the history H directly. Compare this with the model-based definition of the notion of transaction in Section 2.2.

5.2.2 A clean event order

The partial event order \prec in H is a bit too weak for our labeling needs. Our analysis in 2.4 shows that this ordering is compatible with event views: a high-view event cannot precede a low-view event. But it does not have to succeed it either. Also, given the fact that all the view change notification events of a single view can be viewed as occurring at the same time, it would be convenient to collapse them into a single event. This is exactly what we will do now.

Definition 19. *The **Clean Event Set** $(\dot{\mathbb{E}}, \dot{\prec})$ is the partially ordered set obtained from the set \mathbb{E} of events in H by collapsing all the notification events of each view i into a single element ℓ_i , with the induced partial order. Formally:*

$$\dot{\mathbb{E}} = \{\ell_i\}_{0 \leq i < \mathfrak{V}+1} \cup \left(\bigcup_{0 \leq i < \mathfrak{V}+1} (K_i \setminus G_i) \right)$$

$$\dot{e} \dot{\prec} \dot{f} \iff \begin{cases} \dot{e} = \ell_i \text{ and } \dot{f} = \ell_j & \text{and } i < j \\ \dot{e} = \ell_i \text{ and } \dot{f} \in K_j \setminus G_j & \text{and } i \leq j \\ \dot{e} \in K_i \setminus G_i \text{ and } \dot{f} = \ell_j & \text{and } i < j \\ \dot{e} \in K_i \setminus G_i \text{ and } \dot{f} \in K_j \setminus G_j & \text{and } i < j \\ \dot{e}, \dot{f} \in K_i \setminus G_i & \text{and } \dot{e} \prec \dot{f} \end{cases}$$

Notice that since notification events are necessarily trigger events, the elements of $\dot{\mathbb{E}}$ can be divided into trigger events (which include all the ℓ_i events) and side-effect events.

The transactions that are triggered by the critical donation and co-donation packets have a special place in our analysis, so we add some specific notation for them.

Definition 20.

- Let $d \in \overrightarrow{PG^H}$ be the critical donation packet sent by P . We denote $\text{Crit}(P \rightarrow G) = d^{\text{PR}}$ when d^{PR} exists.
- Let $c \in \overrightarrow{GP^H}$ be the critical co-donation packet sent to P . We denote $\text{Crit}(G \rightarrow P) = c^{\text{PR}}$ When c^{PR} exists.

5.2.3 Donation and co-donation sub-transactions

The main concern of the ReceiveDonation and ReceiveCoDonation procedures is the execution of a sequence of ReceiveMessage and ReceiveAck calls. These calls or *sub-transactions* compensate for race conditions. When a new process joins, it is not possible to deliver to it an up-to-date consistent copy of ReplicatedData that will allow it to participate in the distributed computation without any further correction. The problem is that at the moment where the new process joins, some of the information required by it is trapped in packets that are in transit, carrying information that will arrive to the packet's destination too late to be of help.

What are these missing packets? the answer to this question is not easy. Naïvely, most of these are packets that were sent to or from the parent of the new process and failed to arrive by the moment of the view change, but the general picture is a lot more complicated because the parent itself may have joined the group in a previous view change, and the packet of interest may have been sent to/from an ancestor of the parent. Luckily, we only have to worry about the critical view change, which is the first process join in the history. This case, while not trivial, is considerably simpler. Our event labeling will reflect this fact, by labeling the sub-transactions of critical donations and co-donation transactions differently from non-critical ones. This way we can avoid an in-depth analysis of non-critical donations and co-donations, and let their meaning remain entangled in the magic of induction.

In the proceeding discussion we will use the shorthand notation

$$\|\text{vect}\| = \text{vect}.f + \text{vect}.b$$

For vectors like *mpkt_out*, *mpkt_in*[] and index.

Lemma 16. *Let P be an original process (a member of view zero in H) and suppose that G processes a critical donation packet from P . Then there is a one-to-one, order preserving correspondence between the following two sequences:*

1. *The sequence of ReceiveMessage invocations by G at labeled step 1 of the ReceiveDonation procedure as it processes the donation packet from P .*
2. *The queuing events of untimely message packets in channel \overrightarrow{PD} , ordered by \prec . Formally these are the events $k^{\text{QU}} \in \mathbb{E}$ that meet the following criteria:*

$$\begin{aligned} k &\in \overrightarrow{PD} \\ k &= p_{\text{MSG}} \langle \text{msg} \rangle \\ k^{\text{QU}} &\prec v_{\text{crit}}(P)^{\text{PR}} \\ v_{\text{crit}}(D)^{\text{PR}} &\prec k^{\text{PR}} \quad \text{if } k^{\text{PR}} \text{ exists} \end{aligned}$$

Proof. Let k be a packet that meets all the criteria of (2). Then it is a message packet $k = \mathbf{p}_{\text{MSG}}\langle \text{msg} \rangle$ that had been sent from P to D prior to the critical view change. When k is queued, P places a record $\langle \text{msg}, \text{index}, \text{iset} \rangle$ in its `WaitSet`. Since the packet does not make it to D prior to the critical moment, the message does not stabilize with respect to D and remains in `WaitSet` with a D instability at the critical moment. Moreover, it follows from Lemma 9 that at the critical moment at D , $\|\text{mpkt_in}[P]\| < \|\text{index}\|$. Indeed we can say more than that: if k is the i^{th} packet in the sequence of untimely message packets in the \overrightarrow{PD} channel, then $\|\text{mpkt_in}[P]\| + i = \|\text{index}\|$.

Process G starts life with the same value of $\text{mpkt_in}[P]$, inherited from D . Since the `protRun` procedure does not touch this value, and since the critical donation packet from P is the first packet that G receives from P , this value remains unchanged until G executes the `ReceiveDonation` procedure.

At the critical moment P executes the `protJoin` procedure. It adds G instability to the record, with $\text{iset}[G].f = \text{iset}[D].f$ and $\text{iset}[G].b = 0$. P then donates its state to G . When G processes the donation by invoking the `ReceiveDonation` procedure, it finds the record with its G -instability and copies it to the UNT_p set. G sorts the records in such a way that the records of untimely message packets from P are sorted in their broadcast order.

To see why that is true, let k and k' be untimely message packets in \overrightarrow{PD} and suppose that $k'^{\text{QU}} \succ k^{\text{QU}}$. Then $(\text{iset}[D].f)_{@k'} \geq (\text{iset}[D].f)_{@k}$ because this value is initialized from $\text{mpkt_in}[D].f$ (see the `protBroadcast` and `protRemove` procedures) which is monotone increasing. As a result $(\text{iset}[G].f)_{@k'} \geq (\text{iset}[G].f)_{@k}$ in the donated state from P and so

$$\text{height}_1(\langle \text{msg}', \text{index}', \text{iset}' \rangle) \geq \text{height}_1(\langle \text{msg}, \text{index}, \text{iset} \rangle)$$

If these heights are equal the sorting proceeds based on height_2 . This function is derived from the `index` component of the record, which is strictly increasing with each message broadcast at P . Therefore

$$\text{height}_2(\langle \text{msg}', \text{index}', \text{iset}' \rangle) > \text{height}_2(\langle \text{msg}, \text{index}, \text{iset} \rangle)$$

which proves that the records of the untimely message packets are sorted in the order at which those message packets were queued by P .

Process G examines the records in the sorted UNT set one by one. Every time a record from UNT_p is examined, its $\|\text{index}\|$ is compared to $\|\text{mpkt_in}[P]\|$. If $\|\text{index}\|$ is bigger, the `ReceiveMessage` procedure is invoked, causing $\|\text{mpkt_in}[P]\|$ to be incremented by 1. It follows from Lemma 9 that if the message is timely then the comparison will fail and the call will not be invoked. So the call is invoked only for untimely message packets, in the right order, and causes $\|\text{mpkt_in}[P]\|$ to be incremented each time. If the call is invoked for all the untimely message packets up to the i^{th} one, then $\|\text{mpkt_in}[P]\|$ grows by $i - 1$ up to that point, which is not enough to prevent the comparison from succeeding for the i^{th} message packet. Therefore the call is invoked exactly for the records of untimely message packets, in the correct order, as claimed. \square

Lemma 17. *Let P be an original process (a member of view zero in H) and suppose that G processes a critical donation packet from P . Then there is a one-to-one, order preserving correspondence between the following two sequences:*

1. *The sequence of `ReceiveAck` invocations by G at labeled step 2 of the `ReceiveDonation` procedure as it processes the donation packet from P .*

2. The queuing events of untimely forwarded acknowledgement packets in channel \overrightarrow{PD} , ordered by \prec . Formally these are the events $k^{\text{QU}} \in \mathbb{E}$ that meet the following criteria:

$$\begin{aligned}
& k \in \overrightarrow{PD} \\
& k = \mathbf{p}_{\text{ACK}}\langle \text{msg} \rangle \\
& \text{ORIG}(\text{msg}) \neq D \\
& k^{\text{QU}} \prec v_{\text{crit}}(P)^{\text{PR}} \\
& v_{\text{crit}}(D)^{\text{PR}} \prec k^{\text{PR}} \quad \text{if } k^{\text{PR}} \text{ exists}
\end{aligned}$$

Proof. Let k be a packet that meets all the criteria of (2). Then this packet had been sent from P to D prior to the critical view change, in reaction to the receipt of a message msg from D . The message must be forwarded because it did not originate with D . When D sends the message packet to P , it places a record $\langle \text{msg}, \text{index}, \text{iset} \rangle$ in its FwdWaitSet (see labeled step 6 of the protRemove procedure). Let $F_{\text{msg}} = \text{index}.f$.

As the critical parent, D is an original process and Lemma 9 applies. Therefore after P executes the ReceiveMessage procedure, its value of $\text{mpkt_in}[D].f$ is equal to F_{msg} . Let F_{crit} be the value of $\text{mpkt_in}[D].f$ at P at the critical moment. Since the receipt of the message msg at P takes place pre-critically, $F_{\text{crit}} \geq F_{\text{msg}}$. At the critical time P copies $\text{mpkt_in}[D].f$ to $\text{mpkt_in}[G].f$ (while zeroing out the $.b$ component) before sending its donation to G (see the protJoin procedure).

Since the acknowledgement packet for this message does not arrive by the critical moment, the record remains with P in its instability set (iset) and G starts life with the same record in its own FwdWaitSet , after inheriting it from D . Right away G zeroes out $\text{index}.b$ in the record (see the protRun procedure). Since the donation packet is the first packet ever received at G from P , the record must remain in FwdWaitSet at G until that moment, because there is no way until that time for the P -instability to be resolved.

Following the execution of the ReceiveDonation procedure the record of msg in FwdWaitSet makes it into the UNT_g set. Moreover, the value of the $\text{index}.f = F_{\text{msg}}$ component of the record is no higher than the critical value of $\text{mpkt_in}[G].f = F_{\text{crit}}$ at P , which is exactly the value that arrives at G together with P 's donation. The $.b$ field in both values is zero. Therefore the ReceiveAck procedure gets invoked on behalf of msg .

If k' is a packet that meets the same criteria and has a later queuing event than k , then $k' = \mathbf{p}_{\text{ACK}}\langle \text{msg}' \rangle$ where msg' is sent from D to P after msg is sent. Therefore the value of $\|\text{index}\|$ for msg' is higher and therefore the related sub-transaction is executed later, so the mapping in this direction is order preserving.

The inverse correspondence is constructed in a similar way. Let $\langle \text{msg}, \text{index}, \text{iset} \rangle$ Be a record in UNT_g that passes the comparison test

$$\|\text{index}\| \leq \|\text{donation.mpkt_in}[G]\|$$

From the construction of UNT_g it follows that $\text{iset}[P]$ exists. Therefore the message msg got into WaitSet after a message packet queuing event that included P in its target set. When and where did this queuing event happen? If it happened at G , then index must be strictly bigger than the initial value of mpkt_out at G . This initial value is equal to zero in its $.b$ component (see the

protRun procedure) while its $.f$ component is equal to the critical value of $mpkt_out.f$ at D . It follows from Lemma 9 that F_{crit} , the critical value of $mpkt_in[D].f$ at P is at most as high as the critical value of $mpkt_out.f$ at D . Therefore $index.b + index.f > F_{crit}$. But as we have already seen, $F_{crit} = donation.mpkt_in[G].f$ and $donation.mpkt_in[G].b = 0$. This is a contradiction. Therefore the queuing event does not happen at G but rather at D , and the WaitSet record in question is inherited by G when it is created. From this it follows that the message did not originate with D . If it did, then the record would go to BcastWaitSet in D , and this part of WaitSet is not inherited by G but instead zeroed out (see the protRun procedure). So $orig(msg) \neq D$.

So the message msg is forwarded from D to P , and is not stabilized with respect to P by the critical moment. Why is it not stabilized? either it does not arrive at P on time (in which case the message packet from D to P is untimely), or else it does arrive on time but the acknowledgement packet from P to D is untimely. If the first case holds then by Lemma 9 $F_{crit} < index.b + index.f$. As we saw, this leads to a contradiction. Therefore the acknowledgement packet $k = p_{ack}(msg)$ that was sent from P to D was untimely. This means that the packet k meets all the criteria listed in the statement of the lemma.

It is obvious that these two correspondences are inverses of each other and so we are done. \square

Corollary 6. *Let P be an original process and suppose that G processes a critical donation packet from P . Then there is a one-to-one, order preserving correspondence between the following two sequences:*

1. *The sequence of ReceiveMessage and ReceiveAck invocations by G at labeled steps 1 and 2 of the ReceiveDonation procedure as it processes the donation packet from P .*
2. *The queuing events of untimely message packets and forwarded acknowledgement packets in channel \overrightarrow{PD} , ordered by \prec .*

Proof. Most of the claim follows directly from Lemmas 16 and 17. But we still have to show that if $k, k' \in \overrightarrow{PD}$ are an untimely message packet and an untimely acknowledgement packet for a forwarded message respectively, then $k^{qu} \prec k'^{qu}$ if and only if the corresponding records in UNT_p and UNT_g are processed in the same order.

Let $k = p_{msg}(msg)$ and $k' = p_{ack}(msg')$. Let $index'$ be the value of $mpkt_out$ in D when it queues $p_{msg}(msg')$. This is the value of the index field of the record of msg' in $WaitSet(D)$.

Suppose first that $k^{qu} \prec k'^{qu}$. Then P queues k before it processes $p_{msg}(msg')$. By Lemma 9, at the time that P queues k , we have $mpkt_in[D].f \leq index'.f$ at P . But in fact the inequality is strict here because the message msg' is forwarded by D , not originated. Because of that it is $mpkt_out.f$ that is incremented when $p_{msg}(msg')$ is queued and $mpkt_in[D].f$ that is incremented when the same packet is processed. Let $r, r' \in UNT$ be the records that correspond to k and k' respectively. We have already seen that $height_1(r) = mpkt_in[D].f$ and $index(r').f = index'.f$. Therefore

$$height_1(r) = mpkt_in[D].f < index'.f \leq height_1(r')$$

which proves one direction of the claim.

To prove the other direction, suppose that $k^{qu} \succ k'^{qu}$. Then P queues k after it processes $p_{msg}(msg')$. By Lemma 9, at the time that P queues k , we have $mpkt_in[D].f \geq index'.f$ and

following the same logic as before we conclude that $\text{height}_1(r) \geq \text{height}_1(r')$. In this case however we cannot guarantee a strict inequality. In case there is equality, however, the order between r and r' is determined by height_2 . Since $r' \in \text{UNT}_g$ we have $\text{height}_2(r') = 0$ while $r \in \text{UNT}_p$ and therefore

$$\text{height}_2(r) = \|\text{index}(r)\| \geq \text{index}' . f > 0 = \text{height}_2(r')$$

where the rightmost inequality is strict because msg' is a forwarded message. \square

Lemma 18. *Let P be an original process (a member of view zero in H) that processes a critical co-donation packet from G . Then there is a one-to-one, order preserving correspondence between the following two sequences:*

1. *The sequence of ReceiveMessage invocations by P at labeled step 1 of the ReceiveCoDonation procedure as it processes the co-donation packet from G .*
2. *The union of the following two sets of events, ordered by \prec :*
 - (a) *The queuing events of untimely forwarded message packets in channel \overrightarrow{DP} . Formally these are the events $k^{\text{QU}} \in \mathbb{E}$ that meet the following criteria:*

$$\begin{aligned} k &\in \overrightarrow{DP} \\ k &= \mathbf{p}_{\text{MSG}} \langle \text{msg} \rangle \\ \text{ORIG}(\text{msg}) &\neq D \\ k^{\text{QU}} &\prec v_{\text{crit}}(D)^{\text{PR}} \\ v_{\text{crit}}(P)^{\text{PR}} &\prec k^{\text{PR}} \quad \text{if } k^{\text{PR}} \text{ exists} \end{aligned}$$

- (b) *The post-critical, pre- P -donation queuing events of message packets in channel \overrightarrow{GG} . Formally these are the events $k^{\text{QU}} \in \mathbb{E}$ that meet the following criteria:*

$$\begin{aligned} k &\in \overrightarrow{GG} \\ k &= \mathbf{p}_{\text{MSG}} \langle \text{msg} \rangle \\ v_{\text{crit}}(G)^{\text{PR}} &\prec k^{\text{QU}} \prec \text{Crit}(P \rightarrow G) \end{aligned}$$

Note: The union of these two sets of events is linearly ordered by \prec because according to the Parent Axiom $v_{\text{crit}}(D)^{\text{PR}} \prec G_{\text{RUN}} = v_{\text{crit}}(G)^{\text{PR}}$.

Proof. Let k be a packet that meets all the criteria of (2a). This packet was sent from D to P prior to the critical view change, containing a forwarded message msg . When D queued the packet it placed a record $\langle \text{msg}, \text{index}, \text{iset} \rangle$ in its FwdWaitSet . Let $F_{\text{msg}} = \text{index}.f$. Let F_{crit} be the value of $\text{mpkt_in}[D].f$ at P at the critical moment. Then the fact that the packet is untimely, together with Lemma 9 imply that $F_{\text{msg}} > F_{\text{crit}}$. We can say more than that: if k is the i^{th} packet in the sequence of untimely forwarded message packets in the \overrightarrow{DP} channel, then $F_{\text{msg}} = F_{\text{crit}} + i$. At the critical moment process P creates $\text{mpkt_in}[G].f = F_{\text{crit}}$. At the critical moment the value of $\text{mpkt_out}.f$ at D is equal to $F_{\text{crit}} + u$ where u is the number of untimely forwarded message packets in the \overrightarrow{DP} channel. As a result G starts life with $\|\text{mpkt_out}\| = F_{\text{crit}} + u$ (the protRun procedure zeroes out $\text{mpkt_out}.b$).

Since the packet k is untimely, the record remains in **FwdWaitSet**, with P in its instability set, until the critical moment. Therefore G starts life with the same record in its own **FwdWaitSet**, and the **protRun** procedure leaves **index.f** untouched. Since G does not receive any packet from P until it receives its donation, the record remains in **FwdWaitSet** at G until $\text{Crit}(P \rightarrow G)$ and therefore it is sent to P as part of the co-donation from G . As a result, when P processes the co-donation from G it finds the record with its P -instability, and places it in UNT_g .

P sorts the records in such a way that the records of untimely message packets from D are sorted in their broadcast order. To see why that is true, let k and k' be untimely forwarded message packets in \overrightarrow{DP} such that $k'^{\text{qu}} \succ k^{\text{qu}}$. Then $\|(\text{iset}[P])_{@k'}\| \geq \|(\text{iset}[P])_{@k}\|$ at D because this value is initialized from $\|\text{mpkt_in}[P]\|$ (see the **protRemove** procedure) which is monotone increasing. The value of **iset** is not changed by the **protRun** procedure so the same relationship holds in G and so this is what P sees in the co-donated state that it receives from G . Therefore

$$\text{height}_1(\langle \text{msg}', \text{index}', \text{iset}'[] \rangle) \geq \text{height}_1(\langle \text{msg}, \text{index}, \text{iset}[] \rangle)$$

If these heights are equal the sorting proceeds based on height_2 . This function is derived from the **index** component of the record. The **protRun** procedure zeroes out **index.b** but does not touch **index.f**. The original value of **index.f** is derived from the value of **mpkt_out.f** at D . Since k is a forwarded message packet, the value of **mpkt_out.f** is increased after D queues k , and therefore $\text{index}' .f > \text{index} .f$. Therefore at G , where the **.b** component is zero for the records of both k and k' we have $\|\text{index}'\| > \|\text{index}\|$ and therefore

$$\text{height}_2(\langle \text{msg}', \text{index}', \text{iset}'[] \rangle) > \text{height}_2(\langle \text{msg}, \text{index}, \text{iset}[] \rangle)$$

which proves that the records of the untimely message packets are sorted in the order at which those message packets were queued by D .

Now let us examine the other subset of packets. Let k be a packet that meets all the criteria of (2b). When G queues k it does not have P in **ContactSet**, because **ContactSet** is initialized in G to contain only itself (see the **protRun** procedure), and P is only added to the set when G processes the critical donation from P . As a result G does not send a message packet to P even though it does create an **iset**[P] entry in the instability set of the message in its **WaitSet** (to see that, check the **protBroadcast** and **protRemove** procedures to see that message packets are only sent to contacted processes, while instability is created to each process that has an entry in the **mpkt_in** vector. According to Lemma 8(3) **mpkt_in** has entries exactly for the live processes.) This makes it impossible for the record $\langle \text{msg}, \text{index}, \text{iset} \rangle$ of the message to stabilize with respect to P by the time that G processes the donation from P , and as a result the record survives in the co-donated state that G sends to P , together with its P -instability. As a result P copies the record into UNT_g .

The value of **index** in the record is initialized from the value of **mpkt_out** at G . As we noted before G starts life with $\|\text{mpkt_out}\| = F_{\text{crit}} + u$ where u is the number of packets meeting the criteria of 2a. Since **mpkt_out** is incremented by G every time it queues message packets, we can conclude that the j^{th} packet that meets the criteria of 2b has a record with $\|\text{index}\| = F_{\text{crit}} + u + j$. In other words if we take the two sequences 2a and 2b as one, then the record of the i^{th} packet in that sequence has $\|\text{index}\| = F_{\text{crit}} + i$.

We want to show that these records are also sorted by P according to their \prec -order with respect to each other and also with respect to the records of the untimely \overrightarrow{DP} packets.

Let k and k' be two packets in \overrightarrow{GG} that meet the 2b criteria, and let k^0 be a packet in \overrightarrow{DP} that meets the 2a criteria. k and k' are post-critical and both have records that retain their P instability. Moreover, $\text{iset}[P]$ is initialized in both cases from $\text{mpkt_in}[P]$ at G , a value that starts out being equal to the critical value of $\text{mpkt_in}[P]$ at D . This value is untouched by the protRun procedure, and remains unchanged until the moment that G processes the donation from P , because there are no incoming message packets from P to G until that time. Therefore $\text{iset}[P] = \text{iset}'[P]$. Similarly, k^0 gets its value of $\text{iset}[P]$ from a pre-critical value of $\text{mpkt_in}[P]$ at D . Therefore

$$\text{height}_1(\langle \text{msg}', \text{index}', \text{iset}'[] \rangle) = \text{height}_1(\langle \text{msg}, \text{index}, \text{iset}[] \rangle) \geq \text{height}_1(\langle \text{msg}^0, \text{index}^0, \text{iset}^0[] \rangle)$$

So the height_1 function does not resolve the order between the records of k and k' , and if it resolves it between k^0 and k it does so in the desired way.

Both k and k' derive their index value from the value of mpkt_out at G , which increases with each message packet queuing event. therefore

$$\text{height}_2(\langle \text{msg}', \text{index}', \text{iset}'[] \rangle) > \text{height}_2(\langle \text{msg}, \text{index}, \text{iset}[] \rangle)$$

and we have the desired order relation in this case. The record for packet k^0 at D derives its index value from a pre-critical value of mpkt_out at D . G inherits mpkt_out from D with the $.b$ component zeroed out. G inherits the k^0 record and similarly zeroes out its $\text{index}.b$ value. Post-critically, mpkt_out at G continues to grow from this initial value with each message broadcast. Therefore

$$\text{height}_2(\langle \text{msg}, \text{index}, \text{iset}[] \rangle) > \text{height}_2(\langle \text{msg}^0, \text{index}^0, \text{iset}^0[] \rangle)$$

and we have the desired order in this case as well.

Process P examines the records in the sorted UNT set one by one. Every time a record from UNT_g is examined, its $\|\text{index}\|$ is compared to $\|\text{mpkt_in}[G]\|$. If $\|\text{index}\|$ is bigger, the ReceiveMessage procedure is invoked, causing $\|\text{mpkt_in}[G]\|$ to be incremented by 1.

There are exactly three types of records in UNT_g . There are records for message packets that were queued by G itself, namely post-critical message packets that meet the criteria of 2b. Then there are records that were inherited from D . These are records of pre-critical message packets. Some of these packets are untimely and meet the criteria of 2a, while others are timely.

It follows from Lemma 9 that if the message is timely then the comparison will fail and the procedure will not be invoked. So the procedure is invoked only for untimely and post-critical message packets, in the right order, and causes $\|\text{mpkt_in}[G]\|$ to be incremented each time. If the procedure is invoked for all the untimely and post-critical message packets up to the i^{th} one, then $\|\text{mpkt_in}[G]\|$ grows by $i - 1$ up to that point, which is not enough to prevent the comparison from succeeding for the i^{th} message packet. Therefore the procedure is invoked exactly for the records of message packets that meet one of the two criteria, in the correct order, as claimed. \square

Lemma 19. *Let P be an original process (a member of view zero in H) that processes a critical co-donation packet from G . Then there is a one-to-one, order preserving correspondence between the following two sequences:*

1. *The sequence of ReceiveAck invocations by P at labeled step 2 of the ReceiveCoDonation procedure as it processes the co-donation packet from G .*

2. The queuing events of untimely acknowledgement packets in channel \overrightarrow{DP} , ordered by \prec . Formally these are the events $k^{\text{QU}} \in \mathbb{E}$ that meet the following criteria:

$$\begin{aligned} k &\in \overrightarrow{DP} \\ k &= \mathbf{p}_{\text{ACK}}\langle \text{msg} \rangle \\ k^{\text{QU}} &\prec v_{\text{crit}}(D)^{\text{PR}} \\ v_{\text{crit}}(P)^{\text{PR}} &\prec k^{\text{PR}} \quad \text{if } k^{\text{PR}} \text{ exists} \end{aligned}$$

Proof. Let I_{crit} be the value of $\text{mpkt_in}[P]$ at D at the critical moment. G inherits $\text{mpkt_in}[P]$ from D . This value is not changed by the `protRun` procedure, and it remains unchanged until the receipt of a donation packet from P , since this is the first packet of any kind that G receives from P . Upon receipt of the donation packet from P , G immediately sends a co-donation packet to P . Therefore when P executes the `ReceiveCoDonation` procedure, we have:

$$\|\text{co_donation.mpkt_in}[P]\| = \|I_{\text{crit}}\|$$

Let k be a packet that meets all the criteria of (2). Then this packet was sent from D to P prior to the critical view change, in reaction to the receipt of a message msg from P . When P sends the message packet to D , it places a record $\langle \text{msg}, \text{index}, \text{iset} \rangle$ in its `WaitSet`. Let $I_{\text{msg}} = \text{index}$.

Since the acknowledgement packet for this message does not arrive by the critical moment, the record remains in `WaitSet` at P , with D in its instability set until then. At that moment, P adds a G instability to the record (see labeled step 1 of the `protJoin` procedure). This additional instability ensures that the record will remain in `WaitSet` until the first packet from G arrives. This packet is the co-donation from G . Therefore when P processes the co-donation packet it discovers the record in its `WaitSet` and copies it into UNT_P .

As the critical parent, D is an original process and Lemma 9 applies. Therefore after D executes the `ReceiveMessage` procedure, its value of $\text{mpkt_in}[P]$ is equal to I_{msg} . Since the receipt of the message msg at D takes place pre-critically, $\|I_{\text{crit}}\| \geq \|I_{\text{msg}}\|$. Therefore:

$$\|\text{co_donation.mpkt_in}[P]\| = \|I_{\text{crit}}\| \geq \|I_{\text{msg}}\| = \|\text{index}\|$$

therefore the record results in the invocation of the `ReceiveAck` procedure at labeled step 2 of the `ReceiveCoDonation` procedure.

If k' is a packet that meets the same criteria and has a later queuing event than k , then $k' = \mathbf{p}_{\text{ACK}}\langle \text{msg}' \rangle$ where msg' is sent from P to D after msg is sent. Therefore the value of $\|\text{index}\|$ for msg' is higher and therefore the related sub-transaction is executed later, so the mapping in this direction is order preserving.

The inverse correspondence is constructed in a similar way. Let $\langle \text{msg}, \text{index}, \text{iset} \rangle$ Be a record in UNT_P that passes the comparison test

$$\|\text{index}\| \leq \|\text{co_donation.mpkt_in}[P]\|$$

From the construction of UNT_P it follows that $\text{iset}[G]$ exists. Therefore the message msg got into `WaitSet` after a message packet queuing event that included a packet being sent to G (post-critically)

or to D (pre-critically). We must eliminate the post-critical case. If the queuing event is post-critical then index must be strictly bigger than the critical value of mpkt_out at P . Call this critical value O_{crit} . It follows from Lemma 9 that $O_{\text{crit}} \geq I_{\text{crit}}$. Therefore

$$\text{index} > \|O_{\text{crit}}\| \geq \|I_{\text{crit}}\| = \|\text{co_donation.mpkt_in}[P]\|$$

contradicting our assumption. It follows that msg must have been sent pre-critically to D . We know that D processed the message packet pre-critically because the relation

$$\|\text{index}\| \leq \|\text{co_donation.mpkt_in}[P]\| = \|I_{\text{crit}}\|$$

implies it according to Lemma 9. It is obvious that these two correspondences are inverses of each other and so we are done. \square

Corollary 7. *Let P be an original process that processes a critical co-donation packet from G . Then there is a one-to-one, order preserving correspondence between the following two sequences:*

1. *The sequence of ReceiveMessage and ReceiveAck invocations by G at labeled steps 1 and 2 of the ReceiveDonation procedure as it processes the donation packet from P .*
2. *The queuing events of untimely forwarded message packets in channel \overrightarrow{PD} , the queuing events of untimely acknowledgement packets in channel \overrightarrow{PD} and the queuing events of post-critical, pre- P -donation message packets in channel \overrightarrow{GG} , ordered by \prec .*

Proof. Most of the claim follows directly from Lemmas 18 and 19. But we still have to show that if $k, k' \in \overrightarrow{DP}$ are an untimely forwarded message packet and an untimely acknowledgement packet respectively, and if $k'' \in \overrightarrow{GG}$ is a pre- P -donation message packet then

- $k^{\text{qu}} \prec k'^{\text{qu}}$ if and only if the corresponding records in UNT_g and UNT_p are processed in the same order.
- The record for k'' in UNT_g is processed after the record for k' in UNT_p .

Let $k = \mathbf{p}_{\text{MSG}}\langle \text{msg} \rangle$ and $k' = \mathbf{p}_{\text{ACK}}\langle \text{msg}' \rangle$. Let index' be the value of mpkt_out in P when it queues $\mathbf{p}_{\text{MSG}}\langle \text{msg}' \rangle$. This is the value of the index field of the record of msg' the $\text{WaitSet}(P)$.

Suppose first that $k^{\text{qu}} \prec k'^{\text{qu}}$. Then D queues k before it processes $\mathbf{p}_{\text{MSG}}\langle \text{msg}' \rangle$. By Lemma 9, at the time that D queues k , we have $\|\text{mpkt_in}[P]\| < \|\text{index}'\|$ at D . Let $r, r' \in \text{UNT}$ be the records that correspond to k and k' respectively. We have already seen that $\text{height}_1(r) = \|\text{mpkt_in}[P]\|$ and $\text{index}(r') = \text{index}'$. Therefore

$$\text{height}_1(r) = \|\text{mpkt_in}[P]\| < \|\text{index}'\| = \text{height}_1(r')$$

which proves one direction of the claim for k and k' .

To prove the other direction, suppose that $k^{\text{qu}} \succ k'^{\text{qu}}$. Then D queues k after it processes $\mathbf{p}_{\text{MSG}}\langle \text{msg}' \rangle$. By Lemma 9, at the time that D queues k , we have $\|\text{mpkt_in}[P]\| \geq \|\text{index}'\|$ and following the same logic as before we conclude that $\text{height}_1(r) \geq \text{height}_1(r')$. In this case however we cannot guarantee a strict inequality. In case there is equality, however, the order between r and r' is determined by height_2 . Since $r' \in \text{UNT}_p$ we have $\text{height}_2(r') = 0$ while $r \in \text{UNT}_g$ and therefore

$$\text{height}_2(r) = \|\text{index}(r)\| > 0 = \text{height}_2(r')$$

We still have to show that the record for k'' is placed later than the record for k' in the sorted order. As we saw before, G inherits the value of $\text{mpkt_in}[P]$ without change, and since D processes $\mathbf{p}_{\text{MSG}}\langle \text{msg}' \rangle$ pre-critically, the critical value of $\|\text{mpkt_in}[P]\|$ at D is at least as high as $\|\text{index}'\|$. Therefore at the moment that G queues k'' we have $\|\text{mpkt_in}[P]\| \geq \|\text{index}'\|$ and therefore $\text{height}_1(r'') \geq \text{height}_1(r')$ where r'' is the record corresponding to k'' . The rest of the argument is the same as the similar argument for $k^{\text{QU}} \succ k'^{\text{QU}}$. \square

Lemma 20. *Let P be an original process (a member of view zero in H) that processes a critical co-donation packet from G , and suppose that the invocation of `TryToInstall` at labeled step 3 of the `ReceiveCoDonation` procedure results in the installation of the pending views at P . Let $v = \text{cur_view} + \text{v_gap}$ be the view height at P at the moment that it processes the critical co-donation. Then G must have queued a $\mathbf{p}_{\text{FLUSH}}\langle v \rangle$ packet prior to dequeuing the donation from P . Moreover, after queuing that packet and until the donation from P is dequeued, G does not queue any ghost, flush or message packets.*

Proof. The proof of Lemma 15 shows that if `TryToInstall` actually installs the pending views, then the value of flush_height at G must have been equal to v at the time that G processed the donation from P . Let $v_0 = \text{flush_height}_{G@v_{\text{crit}}(G)\text{PR}}$ be the initial flush height at G (see Definition 17). v_0 is equal to $\text{ghost_height}_{D@v_{\text{crit}}(D)\text{PR}}$ and therefore $v_0 < v_{\text{crit}}$ according to Lemma 8(1 and 8). Lemma 8(1) also implies that $v \geq v_{\text{crit}}$ because the co-donation is processed at P after the critical view change. This implies that flush_height at G increases between the time G is created and the time that it processes the donation from P . The only way for that to happen is for G to queue one or more flush packets. The last one of the flush packets before the receipt of the donation from P would reflect the value of flush_height at the time. Therefore this last flush packet must be $k_{\text{last}} = \mathbf{p}_{\text{FLUSH}}\langle v \rangle$. We have to show that G does not queue any additional regular packets between k_{last} and the processing of the donation from P . We already know that G does not queue any additional flush packets in that interval. From Lemma 8(8) and the Piggyback Axiom we know that G also does not queue any ghost packets. It follows from the proof of Lemma 15 that G does not queue any message packets either. \square

5.3 The Label Space

We construct an infinite, very well founded, partially ordered set \mathcal{L} , that we call the *label space*. We use this space as a medium for inducing an ordering on the events in H^r . We do it the following way. First we assign a label from \mathcal{L} to each event in H in a \prec -order preserving manner. Then we expand the assignment to events in H^r . Then we use the latter labeling to induce an order on the events in H^r .

Each label is made up of four coordinates: the constellation coordinate, the sub-transaction coordinate, the adjustment coordinate and the side-effect coordinate. Each coordinate comes from its own very well founded partially ordered set, and the ordering on \mathcal{L} as a whole is left-handed lexicographic, with the constellation coordinate being the major one.

All the events in a single transaction share the same constellation coordinate, but the converse is not true. It is possible for multiple transactions to share the same constellation coordinate. The set of all the transactions that share each constellation coordinate is called a *constellation*. In the

original history H only notification transactions are grouped into constellations that contain more than one transaction. In H^r however this is not the case. We will prove by induction that H and H^r converge. The proof will proceed by induction over constellations (normal induction can be carried over any very well-founded partially ordered set, not just over natural numbers).

5.3.1 The constellation coordinate set

This set is made up of all the trigger events in $\dot{\mathbb{E}}$ with the \prec partial order (see Definition 19).

Definition 21. We use the symbol \mathfrak{L}_c to denote the constellation set. For any transaction T the clean trigger $\text{trig}(T) \in \dot{\mathbb{E}}$ is an element of \mathfrak{L}_c . We denote that element by ℓ_T . For any event $e \in \mathbb{E}^H$ we denote $\ell_e = \ell_{\text{trans}(e)}$. If T is a notification transaction for view i then by definition $\ell_T = \ell_i$.

5.3.2 The sub-transaction coordinate set

This coordinate is only used for differentiating sub-transactions within donation and co-donation transactions. For all other events we use a special zero symbol for this coordinate. These sub-transactions simulate the processing of packets that were supposed to be sent to or from a newly joining process, but that were not sent due to race conditions. For the purpose of constructing H^r the labeling of the critical donation and co-donation sub-transactions is crucial. As for the non-critical donations and co-donations (i.e. those that involve processes other than G that join after the critical view change), their labeling is a lot less important. It would be nice to use the same labeling scheme for all donations and co-donation, and this is indeed possible, but very complicated and not really worth the effort. So reluctantly we use two separate systems of labeling sub-transactions. One for the critical ones and one for the non-critical ones.

The analysis in 5.2.3 shows that the critical sub-transactions in both donation and co-donation transactions can be ordered using related events from $\dot{\mathbb{E}}$. These events are all queuing events of packets, ordered by \prec . This is true for non-critical sub-transactions as well but we have not shown that. So we will simply use positive integers to label those sub-transactions. Together with the additional special zero element this yields:

$$\mathfrak{L}_s = \{\hat{0}\} + \left(\dot{\mathbb{E}} \amalg \mathbb{N}\right)$$

5.3.3 The adjustment coordinate set

This coordinate is used for slight adjustments of trigger events in H^r , forcing them to happen either slightly earlier or slightly later than related triggers. The set contains nine elements:

$$\mathfrak{L}_a = \{\Downarrow^{(-g)} < \Downarrow^{(+g)} < \Downarrow^{(-f)} < \Downarrow^{(+f)} < \Downarrow^{(-s)} < \Downarrow^{(+s)} < \hat{0} < \Uparrow^{(-m)} < \Uparrow^{(+m)}\}$$

Where the elements refer to the time adjustment in processing the following packets:

- $\Downarrow^{(-g)}$: A ghost packet from -G
- $\Downarrow^{(+g)}$: A ghost packet from G
- $\Downarrow^{(-f)}$: A flush packet from -G

- $\Downarrow^{(+f)}$: A flush packet from G
- $\Downarrow^{(-s)}$: An acknowledgement packet from $-G$
- $\Downarrow^{(+s)}$: An acknowledgement packet from G
- $\Uparrow^{(-m)}$: A message packet from $-G$
- $\Uparrow^{(+m)}$: A message packet from G

5.3.4 The side-effect coordinate set

This coordinate is used for differentiating events within a transaction (or a sub-transaction, in the case of a donation or a co-donation that includes multiple sub-transactions). The set contains an infinite sequence of elements

$$\mathfrak{L}_f = \{\hat{0} < \text{CODONATE} < \text{DONATE} < \text{ACK} < \text{GHOST} < \text{FLUSH} < \dots < \text{BCAST}_1 < \text{BCAST}_2 < \dots\}$$

Where $\hat{0}$ indicates a trigger event and the meaning of each of the other symbols is self explanatory.

5.3.5 Labeling the events in H

As we mentioned, each label contains four coordinates, one of each type. Formally

$$\mathfrak{L} = \mathfrak{L}_c \times \mathfrak{L}_s \times \mathfrak{L}_a \times \mathfrak{L}_f$$

If $\ell_T \in \mathfrak{L}_c$, $s \in \mathfrak{L}_s$, $a \in \mathfrak{L}_a$ and $f \in \mathfrak{L}_f$, we use the notation $[\ell_T.s.a|f]$ to denote the label with those coordinates. If T is a critical donation or co-donation transaction we use the notations $[\text{Crit}(G \rightarrow P).s.a|f]$ and $[\text{Crit}(P \rightarrow G).s.a|f]$.

We want to create a map $\lambda : \mathbb{E}^H \rightarrow \mathfrak{L}$ that assigns a label to each event in H and preserves order in the sense that if $e \prec f$ then $\lambda(e) < \lambda(f)$.

Let e be any event in H . We create the label of $e = [\ell_T.s.a|f]$ coordinate by coordinate:

The constellation coordinate:

We simply use the constellation $\ell_{\text{trans}(e)}$.

The sub-transaction coordinate:

- If $\ell_{\text{trans}(e)} = \text{Crit}(P \rightarrow G)$ and e is a side effect of one of the sub-transactions (labeled steps 1 and 2 of ReceiveDonation) then Corollary 6 implies that there is a corresponding event $f = k^{\text{qu}} \in \mathbb{E}$ where $k \in \overrightarrow{PD}$ is an untimely packet. We use f as the sub-transaction coordinate for e .
- If $\ell_{\text{trans}(e)} = \text{Crit}(G \rightarrow P)$ and e is a side effect of one of the sub-transactions (labeled steps 1 and 2 of ReceiveCoDonation) then Corollary 7 implies that there is a corresponding event $f = k^{\text{qu}} \in \mathbb{E}$ where either $k \in \overrightarrow{DP}$ is an untimely packet or $k \in \overrightarrow{GG}$ is a post-critical, pre- P -donation packet. We use f as the sub-transaction coordinate for e .

- If $\ell_{\text{trans}(e)} = \text{Crit}(G \rightarrow P)$ and e is a side effect of the TryToInstall procedure invocation at labeled step 3 of ReceiveCoDonation then Lemma 20 implies that there is a corresponding event $f = k^{\text{qu}} \in \mathbb{E}$ where $k \in \overrightarrow{GG}$ is the last pre- P -donation flush packet, and $k = \mathbf{p}_{\text{FLUSH}}\langle v \rangle$ where v is equal to the view height $\text{cur_view} + v_gap$ at P at the time that it processes the co-donation. We use f as the sub-transaction coordinate for e .
- If $\ell_{\text{trans}(e)}$ is a non-critical donation or co-donation transaction, and e is a side effect of one of the sub-transactions then we label e with a serial number, with all the side effects of the first sub-transaction receiving a value of 1, the side effects of the second sub-transaction receiving a value of 2, et cetera.
- If e is any other event then we label e with the special value $\hat{0}$. This includes the cases where e is
 - a trigger event.
 - a side effect of a transaction that is neither a donation nor a co-donation.
 - the queuing event of the co-donation packet in a donation transaction.
 - a side effect of the TryToInstall invocation in a non-critical co-donation transaction.

The adjustment coordinate:

In H this coordinate is always $\hat{0}$. We only make non-trivial adjustments for events that are added in H^r .

The side-effect coordinate:

We assign this coordinate according to the side-effect. If e is a trigger event, we use the special zero value $\hat{0}$. Otherwise $e = k^{\text{qu}}$ for some packet. In that case we look at that side effect that produced k :

- If $k = \mathbf{p}_{\text{CO-DONATE}}\langle \text{co_donation} \rangle$ then we assign the value CODONATE.
- If $k = \mathbf{p}_{\text{DONATE}}\langle \text{donation} \rangle$ then we assign the value DONATE.
- If $k = \mathbf{p}_{\text{ACK}}\langle \text{msg} \rangle$ then we assign the value ACK.
- If $k = \mathbf{p}_{\text{GHOST}}\langle v \rangle$ then we assign the value GHOST.
- If $k = \mathbf{p}_{\text{FLUSH}}\langle v \rangle$ then we assign the value FLUSH.
- If $k = \mathbf{p}_{\text{MSG}}\langle \text{msg} \rangle$ then we have to differentiate between several cases:
 - If k is an original message packet, in the context of a message broadcast request transaction, we assign the value BCAST_1 .
 - If k is an original message packet $\mathbf{p}_{\text{MSG}}\langle \text{msg} \rangle$ in the context of a flush packet processing transaction (see Lemma 14), we assign the value BCAST_i if msg is the i^{th} original message out of LaunchQueue.
 - If k is an original message packet $\mathbf{p}_{\text{MSG}}\langle \text{msg} \rangle$ in the context of a co-donation packet processing transaction (see Lemma 15), we assign the value BCAST_i if msg is the i^{th} original message out of LaunchQueue.

- If k is a forwarded message packet $\mathbf{p}_{\text{MSG}}\langle \text{msg} \rangle$ in the context of a notification transaction (removal of process P), we assign the value BCAST_i if msg is the i^{th} forwarded message out of $\text{FwdQueue}[P]$

5.3.6 Some labeling examples

Suppose that the application at some process P in H decides to broadcast a message. This decision constitutes a message broadcast request event e_t that results in the invocation of the `protBroadcast` procedure. If $v_gap = 0$ the message gets broadcast, which means that a set of message packets

$$k_{P_1}, k_{P_2}, \dots, k_{P_n}$$

one packet per contacted process, are queued to their respective channels. All of these packets share a single queuing event e_q :

$$k_{P_1}^{\text{qu}} = k_{P_2}^{\text{qu}} = \dots = k_{P_n}^{\text{qu}} = e_q$$

If $v_gap > 0$ then there is no queuing event. Therefore we have a transaction

$$T = \text{trans}_{e_t} = \begin{cases} \{e_t \prec e_q\} & \text{if } v_gap = 0 \\ \{e_t\} & \text{if } v_gap > 0 \end{cases}$$

And the labeling yields

$$\begin{aligned} \lambda(e_t) &= [\ell_T.\hat{0}.\hat{0}|\hat{0}] \\ \lambda(e_q) &= [\ell_T.\hat{0}.\hat{0}|\text{BCAST}_1] \end{aligned}$$

In another example, suppose that the critical joining process G processes a donation from P . The donated state from P includes a record $\langle \text{msg}, \text{index}, \text{iset} \rangle$ in `WaitSet`. This record was placed in `WaitSet` by P when it sent an untimely message packet $k_m = \mathbf{p}_{\text{MSG}}\langle \text{msg} \rangle$ to D . Suppose that the record causes G to invoke the `ReceiveMessage` procedure. When G executes that call it queues an acknowledgement packet $k_s = \mathbf{p}_{\text{ACK}}\langle \text{msg} \rangle$ to P . The queuing event k_s^{qu} is labeled

$$\lambda(k_s^{\text{qu}}) = [\text{Crit}(P \rightarrow G).k_m^{\text{qu}}.\hat{0}|\text{ACK}]$$

Theorem 4. *The map $\lambda : \mathbb{E}^H \rightarrow \mathfrak{L}$ is order preserving*

Proof. Let $e_1, e_2 \in \mathbb{E}^H$ be any events such that $e_1 \prec e_2$. Let $\lambda(e_1) = [\ell_{e_1}.s_{e_1}.\hat{0}|f_{e_1}]$ and $\lambda(e_2) = [\ell_{e_2}.s_{e_2}.\hat{0}|f_{e_2}]$. Let $e_1 \in \mathbb{E}_{P_1}$ and $e_2 \in \mathbb{E}_{P_2}$.

If $P_1 \neq P_2$ then by the minimality of \prec (Minimal Order Axiom) there must be a trigger event e_t at P_2 such that $e_1 \prec e_t \preceq e_2$. The events at P_2 are completely ordered by \prec (Process Order Axiom). It follows from Definition 16 that $e_t \preceq \text{trig}(e_2)$. By the same definition $\text{trig}(e_1) \preceq e_1$. It follows that $\text{trig}(e_1) \prec \text{trig}(e_2)$ and therefore $\ell_{e_1} \prec \ell_{e_2}$. Since λ is left lexicographic it follows that $\lambda(e_1) < \lambda(e_2)$ and we are done in this case.

If $P_1 = P_2$ then it follows from Definition 16 that $\text{trig}(e_1) \preceq \text{trig}(e_2)$ and $\ell_{e_1} \preceq \ell_{e_2}$. If the inequality is strict then we are done. So assume that $\ell_{e_1} = \ell_{e_2}$. This implies that e_1 and e_2 belong to the same transaction T at process P_1 .

We will go over each transaction type and verify that the events in the transaction are labeled with monotonically increasing labels. We rely on the analysis of side effects (see 4.2.1 - 4.2.5). We will denote by e_t the trigger of T , and denote by $e_{s_1}, e_{s_2}, \dots, e_{s_n}$ the side effects of T . Remember throughout the following analysis that $\lambda(e_t) = [\ell_T.\hat{0}.\hat{0}|\hat{0}]$.

Message broadcast request transaction

According to Lemma 11 an application transaction T has one of the following forms:

- No side effects: $T = \{e_t\}$ and there is nothing to prove.
- A message packet multicast: $T = \{e_t \prec e_{s_1}\}$ where $e_{s_1} = \mathbf{p}_{\text{MSG}}\langle \text{msg} \rangle^{\text{qu}}$

$$[\ell_T.\hat{0}.\hat{0}|\hat{0}] < [\ell_T.\hat{0}.\hat{0}|\text{BCAST}_1]$$

Notification transaction

According to Lemma 12 a notification transaction T has one of the following forms:

- No side effects: $T = \{e_t\}$ and there is nothing to prove.
- A ghost packet multicast: $T = \{e_t \prec e_{s_1}\}$ where $e_{s_1} = \mathbf{p}_{\text{GHOST}}\langle v \rangle^{\text{qu}}$

$$[\ell_T.\hat{0}.\hat{0}|\hat{0}] < [\ell_T.\hat{0}.\hat{0}|\text{GHOST}]$$

- A ghost packet multicast followed by a flush packet multicast: $T = \{e_t \prec e_{s_1} \prec e_{s_2}\}$ where $e_{s_1} = \mathbf{p}_{\text{GHOST}}\langle v \rangle^{\text{qu}}$ and $e_{s_2} = \mathbf{p}_{\text{FLUSH}}\langle v \rangle^{\text{qu}}$

$$[\ell_T.\hat{0}.\hat{0}|\hat{0}] < [\ell_T.\hat{0}.\hat{0}|\text{GHOST}] < [\ell_T.\hat{0}.\hat{0}|\text{FLUSH}]$$

- A sequence of message packet multicasts: $T = \{e_t \prec e_{s_1} \prec \dots \prec e_{s_n}\}$ where $e_{s_i} = \mathbf{p}_{\text{MSG}}\langle \text{msg}_i \rangle^{\text{qu}}$

$$[\ell_T.\hat{0}.\hat{0}|\hat{0}] < [\ell_T.\hat{0}.\hat{0}|\text{BCAST}_1] < \dots < [\ell_T.\hat{0}.\hat{0}|\text{BCAST}_n]$$

Message packet processing transaction

According to Lemma 13 a message packet processing transaction T has the form $T = \{e_t, e_{s_1}\}$ where $e_{s_1} = \mathbf{p}_{\text{ACK}}\langle \text{msg} \rangle^{\text{qu}}$

$$[\ell_T.\hat{0}.\hat{0}|\hat{0}] < [\ell_T.\hat{0}.\hat{0}|\text{ACK}]$$

Acknowledgement packet processing transaction

According to Lemma 13 an acknowledgement packet processing transaction T has one of the following forms:

- No side effects.
- A ghost packet multicast.
- A flush packet multicast.
- A ghost packet multicast followed by a flush packet multicast.

The monotonicity of the labeling here works just as in the notification transaction case.

Ghost packet processing transaction

According to Lemma 14 a ghost packet processing transaction T has no side effect so there is nothing to prove in this case.

Flush packet processing transaction

According to Lemma 14 a flush packet processing transaction T has one of the following forms:

- No side effects: $T = \{e_t\}$ and there is nothing to prove.
- A sequence of message packet multicasts: $T = \{e_t \prec e_{s_1} \prec \dots \prec e_{s_n}\}$ where $e_{s_i} = \mathbf{p}_{\text{MSG}} \langle \text{msg}_i \rangle^{\text{qu}}$

$$[\ell_T.\hat{0}.\hat{0}|\hat{0}] < [\ell_T.\hat{0}.\hat{0}|\text{BCAST}_1] < \dots < [\ell_T.\hat{0}.\hat{0}|\text{BCAST}_n]$$

Donation packet processing transaction

According to Lemma 15 a donation packet processing transaction T has the form of a co-donation queuing event followed by a sequence of zero or more sub-transactions

$$T = \{e_t, e_c, e_{s_{1,1}}, \dots, e_{s_{1,k_1}}, e_{s_{2,1}}, \dots, e_{s_{2,k_2}}, \dots, e_{s_{n,1}}, \dots, e_{s_{n,k_n}}\}$$

where e_c is the queuing event of the co-donation packet and $e_{s_{i,1}}, \dots, e_{s_{i,k_i}}$ are the side effects of the i^{th} sub-transaction.

Within each sub-transaction the sub-transaction coordinate of the label is the same for all side effects. Let

$$\lambda(e_{i,j}) = [\ell_T.s_i.\hat{0}|f_{i,j}]$$

Then it follows from the previous cases that for each i the labels are monotonic:

$$[\ell_T.s_i.\hat{0}|f_{i,1}] < [\ell_T.s_i.\hat{0}|f_{i,2}] < \dots < [\ell_T.s_i.\hat{0}|f_{i,k_i}]$$

where s_i is the sub-transaction coordinate of the i^{th} sub-transaction. If the donation is not critical then $s_i = i$, and therefore $s_i < s_{i+1}$ and the whole sequence of sub-transactions is monotonically labeled. If the donation is critical then $s_i = k_i^{\text{qu}}$ where k_i is the untimely packet that is related to the i^{th} sub-transaction by Corollary 6. The same corollary guarantees that if $i < j$ then $k_i^{\text{qu}} \prec k_j^{\text{qu}}$. Therefore in this case as well the whole sequence of sub-transactions is monotonically labeled. In addition

$$\lambda(e_t) = [\ell_T.\hat{0}.\hat{0}|\hat{0}] < [\ell_T.\hat{0}.\hat{0}|\text{CODONATE}] = \lambda(e_c)$$

and both packets have a sub-transaction coordinate that is equal to $\hat{0}$. Therefore the whole donation transaction is labeled monotonically in all cases.

Co-donation packet processing transaction

According to Lemma 15 a co-donation packet processing transaction T has one of the following forms:

- A sequence of zero or more sub-transactions

$$T = \{e_t, e_{s_{1,1}}, \dots, e_{s_{1,k_1}}, e_{s_{2,1}}, \dots, e_{s_{2,k_2}}, \dots, e_{s_{n,1}}, \dots, e_{s_{n,k_n}}\}$$

where $e_{s_{i,1}}, \dots, e_{s_{i,k_i}}$ are the side effects of the i^{th} sub-transaction. This case is resolved the same way as a donation transaction, with Corollary 7 guaranteeing that the order of the sub-transaction coordinates is correct in the critical case.

- A sequence of one or more message packet multicasts: $T = \{e_t \prec e_{s_1} \prec \dots \prec e_{s_n}\}$ where $e_{s_i} = \mathbf{p}_{\text{MSG}} \langle \text{msg}_i \rangle^{\text{qu}}$. If the co-donation is non-critical the labeling looks as follows

$$[\ell_T.\hat{0}.\hat{0}|\hat{0}] < [\ell_T.\hat{0}.\hat{0}|\text{BCAST}_1] < \dots < [\ell_T.\hat{0}.\hat{0}|\text{BCAST}_n]$$

If the co-donation is critical then according to Lemma 20 there is a special post-critical flush packet queuing event $f \in \mathbb{E}$ and the labeling is

$$[\ell_T.\hat{0}.\hat{0}|\hat{0}] < [\ell_T.f.\hat{0}|\text{BCAST}_1] < \dots < [\ell_T.f.\hat{0}|\text{BCAST}_n]$$

□

5.4 Defining The History Reduction Mapping

5.4.1 Preliminaries

We now show how to construct a history H^r that carries the same computation as H but where the critical join event of G is replaced by a removal event of a different process -G. The interesting part is the construction of \mathbb{E}^{H^r} which will rely on the label space that we constructed earlier. We start with the construction of most of the other components of H^r .

$$\begin{aligned} \mathbb{P}^{H^r} &= \mathbb{P}^H \cup \{-G\} \\ \mathbb{P}_h^{H^r} &= \mathbb{P}_h^H \cup \{-G\} \\ \mathfrak{V}^{H^r} &= \mathfrak{V}^H \\ \mathbb{S}_i^{H^r} &= \begin{cases} \mathbb{S}_i^H & \text{if } i \geq v_{\text{crit}} \\ \mathbb{S}_i^H \cup \{G, -G\} & \text{if } i < v_{\text{crit}} \end{cases} \\ \mathbb{A}^{H^r} &= \mathbb{A}^H \end{aligned}$$

The main task is constructing the packets and related events on channels that involve the processes G and -G. We build up these channels starting with the history H as a base, and then adding and removing packets as necessary in an attempt to simulate what would have happened if G and -G were original members of the group and just happened, by some rare chance, to proceed (almost) exactly as the parent D would up to the point of the critical view change, after which -G is removed and G remains to evolve as it originally did in the history H .

The rest of this section is dedicated to the construction of the channels that involve G and -G. The construction involves a detailed description of the packets that are added to these channels, as well as the few packets that are removed. Together with each packet we add its associated events and their labels. In addition we describe the requisite changes to and additions of notification events. This construction amounts to a description of most of the missing ingredients in H^r including all the channels and all the events. To complete the definition of H^r we just need to add in the partial order on the events.

We construct the events in H^r with the goal of simulating a reality where the following things happen:

- G and $-G$ are group members from the start. However, their upper layer application does not get started until the critical view change. Therefore G does not originate any message broadcasts up to the critical point, and $-G$ never originates any message broadcasts at all.
- Other than message origination (and the related flush packets - see below), G and $-G$ proceed exactly like D up to the critical view change. This happens by luck, as multicasts arrive at G and $-G$ at about the same time that they arrive at D , and multicasts and responses from G and $-G$ arrive at other processes at about the same time that similar multicasts and responses arrive from D .
- Despite their participation in group communications, G and $-G$, by sheer luck, have no effect on the state of the group. This happens because acknowledgements from G and $-G$ arrive too early (just before similar acknowledgements arrive from D) and therefore are never decisive in stabilizing messages; forwarded messages from G and $-G$ arrive too late (just after similar forwarded messages arrive from D) and as a result they are discarded as duplicates; and flush packets from G and $-G$ arrive too early (sometimes long before similar packets arrive from D) to be the deciding factor in view installation decisions at the receiving processes.
- One area where G and $-G$ are allowed to diverge from D in a controlled fashion is with the multicasting of flushes. This is unavoidable because G and $-G$ do not originate messages and therefore their respective wait sets tend to be emptier than the one at D . Therefore G and $-G$ may be forced by the CBCAST protocol to multicast a flush long before D is ready to do so. Managing this difference is the whole purpose of the ghost multicasts. These multicasts do not depend on the instability of original messages and therefore can happen at the same time at G , $-G$ and D . Flush multicasts out of G and $-G$, however, do not happen at the same time that similar multicasts occur at D . Rather they immediately follow each ghost multicast.

5.4.2 Some notation

It should hopefully be intuitive at this point that the simulation is achieved, to a large degree, by adding *clone* packets that mimic on $\pm G$ -bound channels what the original packets did on similar D -bound channels. To accommodate the fact that the new flush packets mimic original ghost packets rather than original flush packets, we also introduce the notion of a *zombie* packet. A zombie packet, as the name implies, is a clone of ghost a packet that comes back to life as a flush packet, but otherwise remains unchanged.

The label space \mathfrak{L} plays a crucial role in ordering the new packet events that are created along with the new clone and zombie packets. Instead of ordering these events directly (which can get very complicated) we label them first. Then we introduce an order on all the events in H^r by inducing it back from the labels.

The central challenge in building H^r is the seam between the pre-critical period and the post-critical period. This problem presents itself in the form of packets that are sent pre-critically, but arrive post-critically (including packets that never arrive). Recall that such packets are called *untimely*. We deal with untimely packets using the donation and co-donation transactions. In H^r , the clones and zombies of untimely packets arrive exactly at the time that the receiving process begins the sub-transaction that simulates the arrival of the original packets (see labeled steps 1 and 2 of the ReceiveDonation procedure and labeled steps 1 and 2 of the ReceiveCoDonation procedure at 3.5).

This makes the dequeuing events of donation and co-donation packets pivotal in the definition of the reduction mapping, warranting a specific notation that we introduced in Definition 20 to describe them.

Definition 22. Let msg be a stamped message in H . The **reduction** of msg is an identical message msg^r , but with a slightly different stamping

$$\begin{aligned} \text{ORIG}(msg^r) &= \text{ORIG}(msg) \\ \text{VIEW}(msg^r) &= \text{VIEW}(msg) \\ \text{VT}(msg^r) &= \begin{cases} \text{VT}(msg) \cup \{[G] = 0, [-G] = 0\} & \text{VIEW}(msg) < v_{\text{crit}} \\ \text{VT}(msg) & \text{VIEW}(msg) \geq v_{\text{crit}} \end{cases} \end{aligned}$$

For any data structure A in H , the **reduction** of A is an identical data structure A^r where all the messages contained in A or its substructures have been replaced by their reductions.

If A is a data structure in H with reduction A^r and B is a data structure in H^r such that $B = A^r$, we say that B is **equivalent** to A and we denote it by $B \cong A$.

- Definition 23.**
1. A **clone** of a packet k in H is a new packet in H^r , on a different channel, whose type is identical to the type of k and whose content is the reduction of the content $\text{cont}(k)$. A **clone** of an event e in H is a new event in H^r of the same type as e . A clone event can occur at the same process or at a different one.
 2. A **zombie** of a ghost packet is a clone where, unlike a normal clone, the type of the packet changes from ghost to flush. A **zombie** event is a clone of a ghost packet queuing event in H whose type in H^r is a flush packet queuing event.
 3. A **non- G** process is any $P \neq \pm G$.

We now turn to the heart of the construction of H^r , which involves the construction of its channels and events. The bulk of the work is in constructing the channels and the related packet events, and then there is a bit of additional work in constructing the notification events. We proceed in multiple steps. We start with constructing new queuing events and their labels. We continue with channels where the source is $\pm G$ and the target is **non- G** , then channels where the source is **non- G** and the target is $\pm G$, and then the four channels $\pm G \pm G$. We construct new trigger events and their labels in tandem with the channel construction. Then we construct the new notification events and their labels, and finally we construct the \prec^{H^r} order out of the labels.

5.4.3 Constructing the new queuing events

Most of the new queuing events are added to $\pm G$. The only new queuing events that are added to **non- G** processes are the queuing events of acknowledgement packets that are sent in response to forwarded messages from $\pm G$. This should not be a surprise because we deliberately construct H^r in such a way that the additional packets emanating out of $\pm G$ are largely ignored and therefore have no side effects to speak of.

We only create clone and zombie events for pre-critical queuing events in H . We are going to add post-critical *packets* to H^r , but their queuing events will all be existing queuing events of existing multicasts. In fact many, but not all, of the pre-critical cloned packets will become parts of existing multicasts and will not require a new queuing event.

If $e = k^{\text{qu}}$ is a pre-critical queuing event at D in H labeled $\lambda(e) = [\ell_T.\hat{0}.\hat{0}|f]$ we add the following new events in H^r :

1. If k is a forwarded message packet, or an acknowledgement packet in response to any message packet other than a forwarded message from D
 - (a) A clone queuing event $e_c \in \mathbb{E}_G$ with label $\lambda(e_c) = \lambda(e)$
 - (b) A clone queuing event $e_{-c} \in \mathbb{E}_{-G}$ with label $\lambda(e_{-c}) = \lambda(e)$
2. If $k \in \overrightarrow{DD}$ is an acknowledgement packet in response to a forwarded message packet from D
 - (a) Two clone queuing events $e^{\uparrow(+m)}, e^{\uparrow(-m)} \in \mathbb{E}_D$ with labels

$$\lambda(e^{\uparrow(+m)}) = [\ell_T.\hat{0}.\uparrow^{(+m)}|\text{ACK}]$$

$$\lambda(e^{\uparrow(-m)}) = [\ell_T.\hat{0}.\uparrow^{(-m)}|\text{ACK}]$$

- (b) Three clone queuing events $e_c, e_c^{\uparrow(+m)}, e_c^{\uparrow(-m)} \in \mathbb{E}_G$ with labels

$$\lambda(e_c) = \lambda(e)$$

$$\lambda(e_c^{\uparrow(+m)}) = [\ell_T.\hat{0}.\uparrow^{(+m)}|\text{ACK}]$$

$$\lambda(e_c^{\uparrow(-m)}) = [\ell_T.\hat{0}.\uparrow^{(-m)}|\text{ACK}]$$

- (c) Three clone queuing events $e_{-c}, e_{-c}^{\uparrow(+m)}, e_{-c}^{\uparrow(-m)} \in \mathbb{E}_{-G}$ with labels

$$\lambda(e_{-c}) = \lambda(e)$$

$$\lambda(e_{-c}^{\uparrow(+m)}) = [\ell_T.\hat{0}.\uparrow^{(+m)}|\text{ACK}]$$

$$\lambda(e_{-c}^{\uparrow(-m)}) = [\ell_T.\hat{0}.\uparrow^{(-m)}|\text{ACK}]$$

3. If k is a ghost packet and $\lambda(e) = [\ell_T.\hat{0}.\hat{0}|\text{GHOST}]$
 - (a) A clone queuing event $e_c \in \mathbb{E}_G$ with label $\lambda(e_c) = \lambda(e) = [\ell_T.\hat{0}.\hat{0}|\text{GHOST}]$
 - (b) A clone queuing event $e_{-c} \in \mathbb{E}_{-G}$ with label $\lambda(e_{-c}) = \lambda(e) = [\ell_T.\hat{0}.\hat{0}|\text{GHOST}]$
 - (c) A zombie queuing event $e_z \in \mathbb{E}_G$ with label $\lambda(e_z) = [\ell_T.\hat{0}.\hat{0}|\text{FLUSH}]$
 - (d) A zombie queuing event $e_{-z} \in \mathbb{E}_{-G}$ with label $\lambda(e_{-z}) = [\ell_T.\hat{0}.\hat{0}|\text{FLUSH}]$
4. If k is an original message packet or a flush packet, do not create any clones

If $e = k^{\text{qu}}$ is a pre-critical queuing event in H at $P \neq D$ with label $\lambda(e) = [\ell_T.\hat{0}.\hat{0}|\text{ACK}]$ and if $k \in \overrightarrow{PD}$ is an acknowledgement packet, we add the following new events in H^r :

1. If k is an acknowledgement packet in response to a forwarded message packet
 - (a) A clone queuing event $e^{\uparrow(+m)} \in \mathbb{E}_P$ with label $\lambda(e^{\uparrow(+m)}) = [\ell_T.\hat{0}.\uparrow^{(+m)}|\text{ACK}]$
 - (b) A clone queuing event $e^{\uparrow(-m)} \in \mathbb{E}_P$ with label $\lambda(e^{\uparrow(-m)}) = [\ell_T.\hat{0}.\uparrow^{(-m)}|\text{ACK}]$

2. If k is an acknowledgement packet in response to an original message packet, do not create any clones

5.4.4 Constructing the channels \overrightarrow{PQ} for non- G processes P and Q

The original H channels are copied to H^r almost without change. The only change is that each channel is reduced (see Definition 22). This means that the content $\text{cont}(k)$ of each packet k in the channel is replaced with the reduced content $\text{cont}(k)^r$.

5.4.5 Constructing the channels \overrightarrow{PG} and $\overrightarrow{P(-G)}$ for a non- G process P

We start with an empty channel for $\overrightarrow{P(-G)}$ and with the reduction of the original channel $\overrightarrow{PG^H}$ for \overrightarrow{PG} and *remove* the critical donation packet and its events, if they exist. Then we add the following new packets and events:

For any timely packet $k \in \overrightarrow{PD}$

If k is a ghost, flush or message packet

1. A clone packet $c \in \overrightarrow{PG}$ together with $c^{\text{QU}} = k^{\text{QU}}$ and a new c^{PR} event labeled as $\lambda(c^{\text{PR}}) = \lambda(k^{\text{PR}})$
2. A clone packet $c' \in \overrightarrow{P(-G)}$ together with $c'^{\text{QU}} = k^{\text{QU}}$ and a new c'^{PR} events labeled as $\lambda(c'^{\text{PR}}) = \lambda(k^{\text{PR}})$

If k is an acknowledgement packet in response to a forwarded message

1. A clone packet $c \in \overrightarrow{PG}$ together with $c^{\text{QU}} = (k^{\text{QU}})^{\uparrow(+m)}$ and a new c^{PR} event labeled as $\lambda(c^{\text{PR}}) = \lambda(k^{\text{PR}})$
2. A clone packet $c' \in \overrightarrow{P(-G)}$ together with $c'^{\text{QU}} = (k^{\text{QU}})^{\uparrow(-m)}$ and a new c'^{PR} event labeled as $\lambda(c'^{\text{PR}}) = \lambda(k^{\text{PR}})$

If k is an acknowledgement packet in response to an original message do not create any clones.

For any untimely packet $k \in \overrightarrow{PD}$

If k is a ghost, flush or message packet

1. A clone packet $c \in \overrightarrow{PG}$ together with
 - (a) $c^{\text{QU}} = k^{\text{QU}}$
 - (b) If $\text{Crit}(P \rightarrow G)$ exists, a new c^{PR} event labeled as

$$\lambda(c^{\text{PR}}) = [\text{Crit}(P \rightarrow G).k^{\text{QU}}.\hat{0}|\hat{0}]$$

2. A clone packet $c' \in \overrightarrow{P(-G)}$ together with $c'^{\text{QU}} = k^{\text{QU}}$ but no c'^{PR} event

If k is an acknowledgement packet in response to a forwarded message

1. A clone packet $c \in \overrightarrow{PG}$ together with

- (a) $c^{QU} = (k^{QU})^{\uparrow(+m)}$

- (b) If $\text{Crit}(P \rightarrow G)$ exists, a new c^{PR} event labeled as

$$\lambda(c^{PR}) = [\text{Crit}(P \rightarrow G).k^{QU}.\hat{0}|\hat{0}]$$

2. A clone packet $c' \in \overrightarrow{P(-G)}$ together with $c'^{QU} = (k^{QU})^{\uparrow(-m)}$ but no c'^{PR} event

If k is an acknowledgement packet in response to an original message do not create any clones.

5.4.6 Constructing the channels \overrightarrow{GP} and $\overrightarrow{(-G)P}$ for a non- G process P

We start with the empty channel for $\overrightarrow{(-G)P}$ and with the reduction of the original channel $\overrightarrow{GP^H}$ for \overrightarrow{GP} and *remove* the critical co-donation packet and its related events. Then we add the following new packets and events:

For any timely packet $k \in \overrightarrow{DP}$ with $e = k^{QU}$ and $\lambda(k^{PR}) = [\ell_{T'}. \hat{0}.\hat{0}|\hat{0}]$

If k is a forwarded message packet

1. A clone packet $c \in \overrightarrow{GP}$ together with $c^{QU} = e_c$ and a new c^{PR} event labeled as $\lambda(c^{PR}) = [\ell_{T'}. \hat{0}.\uparrow^{(+m)}|\hat{0}]$
2. A clone packet $-c \in \overrightarrow{(-G)P}$ together with $-c^{QU} = e_{-c}$ and a new $-c^{PR}$ event labeled as $\lambda(-c^{PR}) = [\ell_{T'}. \hat{0}.\uparrow^{(-m)}|\hat{0}]$

If k is a ghost packet

1. A clone packet $c \in \overrightarrow{GP}$ together with $c^{QU} = e_c$ and a new c^{PR} event labeled as $\lambda(c^{PR}) = [\ell_{T'}. \hat{0}.\downarrow^{(+g)}|\hat{0}]$
2. A zombie packet $z \in \overrightarrow{GP}$ together with $z^{QU} = e_z$ and a new z^{PR} event labeled as $\lambda(z^{PR}) = [\ell_{T'}. \hat{0}.\downarrow^{(+f)}|\hat{0}]$
3. A clone packet $-c \in \overrightarrow{(-G)P}$ together with $-c^{QU} = e_{-c}$ and a new $-c^{PR}$ event labeled as $\lambda(-c^{PR}) = [\ell_{T'}. \hat{0}.\downarrow^{(-g)}|\hat{0}]$
4. A zombie packet $-z \in \overrightarrow{(-G)P}$ together with $-z^{QU} = e_{-z}$ and a new $-z^{PR}$ event labeled as $\lambda(-z^{PR}) = [\ell_{T'}. \hat{0}.\downarrow^{(-f)}|\hat{0}]$

If k is an acknowledgement packet

1. A clone packet $c \in \overrightarrow{GP}$ together with $c^{QU} = e_c$ and a new c^{PR} event labeled as $\lambda(c^{PR}) = [\ell_{T'}. \hat{0}.\downarrow^{(+s)}|\hat{0}]$
2. A clone packet $-c \in \overrightarrow{(-G)P}$ together with $-c^{QU} = e_{-c}$ and a new $-c^{PR}$ event labeled as $\lambda(-c^{PR}) = [\ell_{T'}. \hat{0}.\downarrow^{(-s)}|\hat{0}]$

If k is a flush or an original message packet do not create any clones.

For any untimely packet $k \in \overrightarrow{D\hat{P}}$ with $e = k^{\text{QU}}$

If k is an acknowledgement packet or a forwarded message packet

1. A clone packet $c \in \overrightarrow{G\hat{P}}$ together with $c^{\text{QU}} = e_c$ and if $\text{Crit}(G \rightarrow P)$ exists, a new c^{PR} event labeled as $\lambda(c^{\text{PR}}) = [\text{Crit}(G \rightarrow P).k^{\text{QU}}.\hat{0}|\hat{0}]$
2. A clone packet $-c \in \overrightarrow{(-G)\hat{P}}$ together a $-c^{\text{QU}} = e_{-c}$ and no $-c^{\text{PR}}$ event

If k is a ghost packet

1. A clone packet $c \in \overrightarrow{G\hat{P}}$ together with $c^{\text{QU}} = e_c$ and if $\text{Crit}(G \rightarrow P)$ exists, a new c^{PR} event labeled as $\lambda(c^{\text{PR}}) = [\text{Crit}(G \rightarrow P).k^{\text{QU}}.\Downarrow^{(+g)}|\hat{0}]$
2. A zombie packet $z \in \overrightarrow{G\hat{P}}$ together with $z^{\text{QU}} = e_z$ and if $\text{Crit}(G \rightarrow P)$ exists, a z^{PR} event labeled as $\lambda(z^{\text{PR}}) = [\text{Crit}(G \rightarrow P).k^{\text{QU}}.\Downarrow^{(+f)}|\hat{0}]$
3. A clone packet $-c \in \overrightarrow{(-G)\hat{P}}$ together with $-c^{\text{QU}} = e_{-c}$ and no $-c^{\text{PR}}$ event
4. A zombie packet $-z \in \overrightarrow{(-G)\hat{P}}$ together with $-z^{\text{QU}} = e_{-z}$ and no $-z^{\text{PR}}$ event

If k is a flush or an original message packet do not create any clones.

For any post-critical packet $k \in \overrightarrow{G\hat{G}}$ with $\lambda(k^{\text{QU}}) = [\ell_T.j.\hat{0}|f]$ where $\text{view}(T) < r(P)$ and, if $\text{Crit}(P \rightarrow G)$ exists, $\ell_T \dot{<} \text{Crit}(P \rightarrow G)$

If k is a ghost, flush or message packet, a clone packet $c \in \overrightarrow{G\hat{P}}$ with $c^{\text{QU}} = k^{\text{QU}}$ and if $\text{Crit}(G \rightarrow P)$ exists, a new c^{PR} event labeled as $\lambda(c^{\text{PR}}) = [\text{Crit}(G \rightarrow P).k^{\text{QU}}.\hat{0}|\hat{0}]$

If k is an acknowledgement, donation or co-donation packet, do not create any clones.

5.4.7 Constructing the channels $\overrightarrow{(\pm G)(\pm G)}$

We start with the empty channels for the three channels involving $-G$ and with the reduction of the original channel $\overrightarrow{G\hat{G}^H}$ for $\overrightarrow{G\hat{G}}$. Then we add the following new packets and events (notice that by the Self Channel Axiom there are no untimely packets in this case):

For any timely packet $k \in \overrightarrow{D\hat{D}}$ with $e = k^{\text{QU}}$ and $\lambda(k^{\text{PR}}) = [\ell_{T'}.\hat{0}.\hat{0}|\hat{0}]$

If k is a forwarded message packet

1. A clone packet $c \in \overrightarrow{G\hat{G}}$ together with $c^{\text{QU}} = e_c$ and a new c^{PR} event labeled as $\lambda(c^{\text{PR}}) = [\ell_{T'}.\hat{0}.\uparrow^{(+m)}|\hat{0}]$
2. A clone packet $c' \in \overrightarrow{G(-G)\hat{G}}$ together with $c'^{\text{QU}} = e_c$ and a new c'^{PR} event labeled as $\lambda(c'^{\text{PR}}) = [\ell_{T'}.\hat{0}.\uparrow^{(+m)}|\hat{0}]$
3. A clone packet $-c \in \overrightarrow{(-G)\hat{G}}$ together with $-c^{\text{QU}} = e_{-c}$ and a new $-c^{\text{PR}}$ event labeled as $\lambda(-c^{\text{PR}}) = [\ell_{T'}.\hat{0}.\uparrow^{(-m)}|\hat{0}]$
4. A clone packet $-c' \in \overrightarrow{(-G)(-G)\hat{G}}$ together with $-c'^{\text{QU}} = e_{-c}$ and a new $-c'^{\text{PR}}$ event labeled as $\lambda(-c'^{\text{PR}}) = [\ell_{T'}.\hat{0}.\uparrow^{(-m)}|\hat{0}]$

If k is an acknowledgement packet for a forwarded message

1. A clone packet $c \in \overrightarrow{GG}$ together with $c^{QU} = e_c^{\uparrow(+m)}$ and a new c^{PR} event labeled as $\lambda(c^{PR}) = [\ell_{T'}. \hat{0}. \downarrow^{(+s)} | \hat{0}]$
2. A clone packet $c' \in \overrightarrow{G(-G)}$ together with $c'^{QU} = e_c^{\uparrow(-m)}$ and a new c'^{PR} event labeled as $\lambda(c'^{PR}) = [\ell_{T'}. \hat{0}. \downarrow^{(+s)} | \hat{0}]$
3. A clone packet $-c \in \overrightarrow{(-G)G}$ together with $-c^{QU} = e_{-c}^{\uparrow(+m)}$ and a new $-c^{PR}$ event labeled as $\lambda(-c^{PR}) = [\ell_{T'}. \hat{0}. \downarrow^{(-s)} | \hat{0}]$
4. A clone packet $-c' \in \overrightarrow{(-G)(-G)}$ together with $-c'^{QU} = e_{-c}^{\uparrow(-m)}$ and a new $-c'^{PR}$ event labeled as $\lambda(-c'^{PR}) = [\ell_{T'}. \hat{0}. \downarrow^{(-s)} | \hat{0}]$

If k is a ghost packet

1. A clone packet $c \in \overrightarrow{GG}$ together with $c^{QU} = e_c$ and a new c^{PR} event labeled as $\lambda(c^{PR}) = [\ell_{T'}. \hat{0}. \downarrow^{(+g)} | \hat{0}]$
2. A zombie packet $z \in \overrightarrow{GG}$ together with $z^{QU} = e_z$ and a new z^{PR} event labeled as $\lambda(z^{PR}) = [\ell_{T'}. \hat{0}. \downarrow^{(+f)} | \hat{0}]$
3. A clone packet $c' \in \overrightarrow{G(-G)}$ together with $c'^{QU} = e_c$ and a new c'^{PR} event labeled as $\lambda(c'^{PR}) = [\ell_{T'}. \hat{0}. \downarrow^{(+g)} | \hat{0}]$
4. A zombie packet $z' \in \overrightarrow{G(-G)}$ together with $z'^{QU} = e_z$ and a new z'^{PR} event labeled as $\lambda(z'^{PR}) = [\ell_{T'}. \hat{0}. \downarrow^{(+f)} | \hat{0}]$
5. A clone packet $-c \in \overrightarrow{(-G)G}$ together with $-c^{QU} = e_{-c}$ and a new $-c^{PR}$ event labeled as $\lambda(-c^{PR}) = [\ell_{T'}. \hat{0}. \downarrow^{(-g)} | \hat{0}]$
6. A zombie packet $-z \in \overrightarrow{(-G)G}$ together with $-z^{QU} = e_{-z}$ and a new $-z^{PR}$ event labeled as $\lambda(-z^{PR}) = [\ell_{T'}. \hat{0}. \downarrow^{(-f)} | \hat{0}]$
7. A clone packet $-c' \in \overrightarrow{(-G)(-G)}$ together with $-c'^{QU} = e_{-c}$ and a new $-c'^{PR}$ event labeled as $\lambda(-c'^{PR}) = [\ell_{T'}. \hat{0}. \downarrow^{(-g)} | \hat{0}]$
8. A zombie packet $-z' \in \overrightarrow{(-G)(-G)}$ together with $-z'^{QU} = e_{-z}$ and a new $-z'^{PR}$ event labeled as $\lambda(-z'^{PR}) = [\ell_{T'}. \hat{0}. \downarrow^{(-f)} | \hat{0}]$

5.4.8 Constructing the notification events

The notification events and their labels are mostly left unchanged from H . We start out with $\mathbb{F}^{H^r} = \mathbb{F}^H$ and then apply the following additions and substitutions:

We add notifications $v_0(G)$ and $v_0(-G)$ to \mathbb{F}^{H^r} with

$$\begin{aligned} \text{cont}(v_0(G)) &= \mathbf{n}_{\text{START}} \langle G \rangle \\ \text{cont}(v_0(-G)) &= \mathbf{n}_{\text{START}} \langle -G \rangle \end{aligned}$$

For any view $0 < i < v_{\text{crit}}$ we add notifications $v_i(G)$ and $v_i(-G)$ to \mathbb{F}^{H^r} with

$$\text{cont}(v_i(G)) = \text{cont}(v_i(-G)) = \text{cont}(v_i(D))$$

For any non- G process P we replace the contents of the critical notification

$$\text{cont}(v_{v_{\text{crit}}}(P)) = \mathbf{n}_{\text{JOIN}}\langle G, D \rangle$$

with

$$\text{cont}(v_{v_{\text{crit}}}(P)) = \mathbf{n}_{\text{REM}}\langle -G \rangle$$

We replace the contents of the critical notification

$$\text{cont}(v_{v_{\text{crit}}}(G)) = \mathbf{n}_{\text{START}}\langle G \rangle$$

with

$$\text{cont}(v_{v_{\text{crit}}}(G)) = \mathbf{n}_{\text{REM}}\langle -G \rangle$$

We add a notification $v_{v_{\text{crit}}}(-G)$ to \mathbb{F}^{H^r} with

$$\text{cont}(v_{v_{\text{crit}}}(-G)) = \mathbf{n}_{\text{STOP}}$$

For any view $0 \leq i < v_{\text{crit}}$ we add notification events $v_i(G)^{\text{PR}}$ and $v_i(-G)^{\text{PR}}$ to \mathbb{E}_G and \mathbb{E}_{-G} respectively, labeled as $\lambda(v_i(G)^{\text{PR}}) = \lambda(v_i(-G)^{\text{PR}}) = [\ell_i.\hat{0}.\hat{0}|\hat{0}]$

5.4.9 The partial order \prec^{H^r} and dropped items

We now construct the order \prec in H^r . We will occasionally denote it by \prec^{H^r} . As we mentioned before, we use the partial order on labels in the construction.

The relation \prec^{H^r} is defined as the transitive closure of the following primitive order relations:

1. For each $k \in \mathbb{K}^{H^r}$:

$$k^{\text{QU}} \prec^{H^r} k^{\text{PR}}$$

whenever the dequeuing event exists.

2. For each process $X \in \mathbb{P}^{H^r}$ and any two events $e_1, e_2 \in \mathbb{E}_X^{H^r}$:

$$e_1 \prec^{H^r} e_2$$

if $\lambda(e_1) < \lambda(e_2)$

3. For any parent/child pair E/J in H , other than the critical pair D/G :

$$v_{j(J)}(E)^{\text{PR}} \prec^{H^r} J_{\text{RUN}}$$

Please note that this definition produces a well defined *relation* \prec^{H^r} , but it will require some work to show that it is actually a weak partial order. We will pursue that investigation a little later.

We need to define for each new packet whether it is sent and received, and for each new notification whether it is dropped. We declare that there are no dropped notifications in H^r . As for packets, the packets for which we added a dequeuing event must be received, and therefore must be sent. But there are packets for which we did not add a dequeuing event. Since we do not need H^r to be transactional or even lossless, but only conforming, we simply declare that any clone or zombie for which there is no dequeuing event is dropped, meaning that it is sent but not received.

5.4.10 Implementing the user application interface

To complete the construction of H^r we have to insulate the user application from any knowledge of the fact that group history has changed. In order to do that we have to change the implementation of the user application interface (see 2.3.3), essentially creating a thin wrapper around it that hides the effects of the history changes. As part of the wrapper we add two new variables, *ul_view* and *ul_count*, to the ReplicatedData structure, and we make use of three "magic numbers" that are defined below. The new variables count how many views were installed and how many messages were delivered so far, in order to discover the boundary between the pre-critical and post-critical time that would otherwise be invisible to the user application.

Definition 24. *MAGIC_VIEW* is the value of *cur_view* at *D* in *H* at the critical time.

MAGIC_MSG is the number of messages that are delivered at *D* in *H* by the critical time.

MAGIC_SIZE is the number of members of view zero in *H*.

We define two different implementations of the interface in H^r , one for $\pm G$ and one for all other processes. As usual, we will use *P* to denote any process which is not $\pm G$. We use the suffixes "*@P*" and "*@G*" to differentiate between the two implementations. We also use the marker "*r*" to indicate the H^r version of an up-call.

Procedure GroundState^r@P

```
let ul_view = - MAGIC_SIZE - 2;
let ul_count = 0;
GroundState@P(); // call the original H version
```

Procedure GroundState^r@G

```
let ul_view = - MAGIC_SIZE - 2;
let ul_count = 0;
GroundState@D(); // call the original H version at process D
```

Procedure ApplyMessage^r@P(msg, originator)

```
ApplyMessage@P(msg, originator);
```

Procedure ApplyMessage^r@G(msg, originator)

```
increment ul_count;  
if ul_count ≤ MAGIC_MSG then  
    ApplyMessage@D(msg, originator); // call the original D version  
end  
else  
    ApplyMessage@G(msg, originator); // call the original G version  
end
```

Procedure ApplyJoin^r@P(pid)

```
increment ul_view;  
if ul_view ≠ 0 and ul_view ≠ -1 then  
    ApplyJoin@P(pid);  
end
```

Procedure ApplyJoin^r@G(pid)

```
increment ul_view;  
if ul_view > MAGIC_VIEW then  
    ApplyJoin@G(pid);  
end  
else if ul_view ≠ 0 and ul_view ≠ -1 then  
    ApplyJoin@D(pid);  
end
```

Procedure ApplyRemoval^r@P(pid)

```
increment ul_view;  
if ul_view =  $v_{crit}$  then  
    ApplyJoin@G(G); // pid is equal to -G  
end  
else  
    ApplyRemoval@P(pid);  
end
```

Procedure ApplyRemoval^r@G(pid)

```

increment  $ul\_view$ ;
if  $ul\_view \leq \text{MAGIC\_VIEW}$  then
    ApplyRemoval@D(pid);
end
else if  $ul\_view = v_{\text{crit}}$  then
    ApplyJoin@G(G); // pid is equal to -G
end
else
    ApplyRemoval@G(pid);
end

```

5.5 Basic Properties Of H^r

Now that we have defined the reduced history H^r , we establish its basic properties. We show that H^r is a history according to Definition 1, and indeed a conforming history according to Definition 7. Only after we do that can we show that H^r is the history of a valid CBCAST execution and that H^r carries the same computation as H .

Definition 25. For any packet $k \in \mathbb{K}^{H^r}$ define

$$\text{origin}(k) = \begin{cases} k & \text{if } k \text{ is the reduction of an original } \mathbb{K}^H \text{ packet} \\ j & \text{if } k \text{ is a clone or zombie of the original packet } j \end{cases}$$

Theorem 5. H^r is a history.

We start with a lemma about the relation \prec^{H^r}

Lemma 21. If e and f are packet events in H^r and $e \prec f$ then $\lambda(e) < \lambda(f)$

Proof. The relation \prec on the packet events in H^r is the transitive closure of the three primitive relations defined in 5.4.9. If we show that each of the primitive relations is compatible with the label order then we are done.

The second primitive relation is compatible with label order by construction, and the third primitive relation is compatible because both notification events share the same label $[\ell_{j(J)}.\hat{0}.\hat{0}|\hat{0}]$ so the only difficulty is showing that for every packet $k' \in \mathbb{K}^{H^r}$ for which k'^{PR} exists, $\lambda(k'^{\text{QU}}) < \lambda(k'^{\text{PR}})$.

If the packet k' is an original H packet, then this property follows from Theorem 4. We only need to verify this property for packets k' that are either clones or zombies of an original packet k . Looking through the construction of H^r packets, one can easily observe that one of the following three possibilities must occur:

- k is a timely packet with $\lambda(k^{\text{QU}}) = [\ell_T.*.*|*]$ and $\lambda(k^{\text{PR}}) = [\ell_{T'}.*.*|*]$ where $\ell_T \dot{\prec} \ell_{T'}$. In that case for any clone or zombie k' of k , $\lambda(k'^{\text{QU}}) = [\ell_T.*.*|*]$ and $\lambda(k'^{\text{PR}}) = [\ell_{T'}.*.*|*]$. Therefore $\lambda(k'^{\text{QU}}) < \lambda(k'^{\text{PR}})$.
- k is an untimely packet with $\lambda(k^{\text{QU}}) = [\ell_T.*.*|*]$ where $\ell_T \dot{\prec} \ell_{v_{\text{crit}}}$. In that case $\lambda(k'^{\text{QU}}) = [\ell_T.*.*|*]$ and if k'^{PR} exists then $\lambda(k'^{\text{PR}}) = [\ell_{T'}.*.*|*]$ where $\ell_{T'} = \text{Crit}(P \rightarrow G)$ or $\ell_{T'} =$

$\text{Crit}(G \rightarrow P)$ for some original process P . In either case $\ell_{T'} \succeq \ell_{v_{\text{crit}}}$ and therefore $\lambda(k'^{\text{QU}}) < \lambda(k'^{\text{PR}})$.

- k is packet in \overrightarrow{GG} with $\lambda(k^{\text{QU}}) = [\ell_T.*.*|*]$ where $\ell_{v_{\text{crit}}} \prec \ell_T \prec \text{Crit}(P \rightarrow G)$ for some original process P . In that case $\lambda(k'^{\text{QU}}) = [\ell_T.*.*|*]$ and $\lambda(k'^{\text{PR}}) = [\text{Crit}(G \rightarrow P).k^{\text{QU}}.\hat{0}|\hat{0}]$. If c is the co-donation packet sent from G to P in the original history H then

$$\begin{aligned} \lambda(k'^{\text{QU}}) &< [\text{Crit}(P \rightarrow G).\hat{0}.\hat{0}|\hat{0}] < [\text{Crit}(P \rightarrow G).\hat{0}.\hat{0}|\text{CODONATE}] = \lambda(c^{\text{QU}}) < \\ &< \lambda(c^{\text{PR}}) = [\text{Crit}(G \rightarrow P).\hat{0}.\hat{0}|\hat{0}] < [\text{Crit}(G \rightarrow P).k^{\text{QU}}.\hat{0}|\hat{0}] = \lambda(k'^{\text{PR}}) \end{aligned}$$

This concludes the proof. \square

Corollary 8. *Let $e, f \in \mathbb{E}_P^{H^r}$ be any two events at a process P in H^r . Then $e \prec f$ if and only if $\lambda(e) < \lambda(f)$.*

Proof. Lemma 21 proves one direction. The other direction follows immediately from the definition of \prec^{H^r} in 5.4.9. \square

Lemma 22. *Let $e \in \mathbb{E}^{H^r}$ be the queuing event of a message, ghost or flush packet k and let T_e be its target set (refer to the Process Order Axiom for the definition of target set). Let $e_0 = \text{origin}(k)^{\text{QU}}$. Then*

1. *if e_0 is a pre-critical event that occurred at a process R then*

$$T_e = \text{ContactSet}_{R@e_0} \cup \{\pm G\}$$

2. *if e_0 is a post-critical event that occurred at G then*

$$T_e = \text{LiveSet}_{G@e_0}$$

3. *if e_0 is a post-critical event that occurred at $R \neq G$ then*

$$T_e = \text{ContactSet}_{R@e_0}$$

Proof. First take a quick look at the pseudo code to verify that every multicast of message, ghost and flush packets has a target set that is equal to **ContactSet**.

The claims follow from a careful examination of the construction of events in H^r .

Let $R \in \mathbb{P}^H$ and $R \neq G$. Post-critically we do not add any new clones or zombies to packets emanating from R so in this case $\text{origin}(k) = k$ and T_e is the same as the original target set, which proves the third part.

Pre-critically we do add clones and zombies to message, ghost and flush packets emanating from R . If $R \neq D$ then all of these clones (there are no zombies) also emanate from R . If $R = D$ then some of the clones and zombies emanate from $\pm G$.

Every pre-critical multicast set of message, ghost or flush packets in R contains exactly one packet that is targeted at D . This packet gets cloned with two new targets $\pm G$ and added to the same multicast set. This proves the first part in this case when k emanates from R .

If $R = D$ then for every forwarded message multicast there is a cloned multicast emanating from $\pm G$ that contains the complete original target set with the addition of $\pm G$. Original message and flush multicasts do not get cloned to $\pm G$. Each ghost multicast is cloned and zombied to two consecutive multicasts, both of which have the required target set. This takes care of the first part.

As for G , to every post-critical multicast we add packets destined to the uncontacted processes. It follows from Lemma 8(2) that this makes the target set equal to `LiveSet`. \square

The following is a technical lemma that is required for proving that the channels in H^r are FIFO.

Lemma 23. *Let X and Y be processes in H^r and let $k_1, k_2 \in \overrightarrow{XY}^{H^r}$ be distinct pre-critical packets such that $k_1^{\text{qu}} \prec k_2^{\text{qu}}$. Then $\text{origin}(k_1)^{\text{qu}} \preceq \text{origin}(k_2)^{\text{qu}}$, with equality occurring exactly when $\text{origin}(k_1) = \text{origin}(k_2)$ is a pre-critical ghost packet and $X = \pm G$.*

Proof. If both k_1 and k_2 are original, there is nothing to prove. The case where one of them is original and one is a clone or zombie is not possible because all the pre-critical clones and zombies belong to channels that involve $\pm G$ while none of the original pre-critical packets do. Therefore we can assume that both k_1 and k_2 are either clones or zombies of their original packets.

Let $\lambda(k_i^{\text{qu}}) = [\ell_{T_i}.a_i.b_i|c_i]$. By Corollary 8 we know that $\lambda(k_1^{\text{qu}}) < \lambda(k_2^{\text{qu}})$ and therefore $\text{trig}(T_1) \preceq \text{trig}(T_2)$. By going through the different cases of 5.4.5, 5.4.6 and 5.4.7 it is easy to verify that for any pre-critical packet k with $\lambda(k^{\text{qu}}) = [\ell_T.a.b|c]$, the event $\text{origin}(k)^{\text{qu}}$ has the same constellation label ℓ_T . We also know that for any event e in H the constellation label is $\ell_{\text{trans}(e)}$ (see 5.3.5). Therefore $\text{trig}(T_i) = \text{trig}(\text{origin}(k_i)^{\text{qu}})$.

If $\text{trig}(T_1) \prec \text{trig}(T_2)$ then by Definition 16 $\text{origin}(k_1)^{\text{qu}} \prec \text{origin}(k_2)^{\text{qu}}$ and we are done. Otherwise $T_1 = T_2$ so $\text{origin}(k_1)$ and $\text{origin}(k_2)$ are queued in the same transaction. Denote $T = T_1 = T_2$. We also know that $\text{origin}(k_1)$ and $\text{origin}(k_2)$ belong to the same channel. This channel is obtained from \overrightarrow{XY} by replacing any occurrence of $\pm G$ with D .

Going over the complete list of different possibilities of transaction content for T we have:

- T comprises a trigger plus one or more multicasts of original or forwarded message packets. This happens if T is an message broadcast request transaction, or if T is a notification transaction that resulted in the forwarding of messages out of `FwdQueue[P]`, or if T is a flush packet transaction that caused the installation of a new view and the broadcasting of one or more messages out of `LaunchQueue`. In these cases $\text{origin}(k_i)$ are both pre-critical message packets. Going over the different cases in 5.4.5 - 5.4.7 shows that in this case $\lambda(k_i^{\text{qu}}) = \lambda(\text{origin}(k_i)^{\text{qu}})$ and the lemma follows.
- T comprises a trigger and the queuing of exactly one acknowledgement packet k . This happens if T is triggered by the processing of a message packet. In this case it must be that $\text{origin}(k_1) = \text{origin}(k_2) = k$. But an examination of 5.4.5 - 5.4.7 shows that distinct clones of pre-critical acknowledgement packets belong to different channels, contrary to the assumption that k_1 and k_2 belong to the same channel. So this case does not occur.
- T comprises a trigger followed by a single multicast of ghost packets. This happens if T is triggered by the processing of an acknowledgement packet that cleared out `FwdWaitSet` while `BcastWaitSet` remained non-empty, or if T is triggered by a $\mathbf{n}_{\text{REM}}\langle P \rangle$ notification event that occurred when `FwdQueue[P]` and `FwdWaitSet` were empty while `BcastWaitSet` was not

empty. In this case there is a single packet at each relevant channel, and therefore $\text{origin}(k_1) = \text{origin}(k_2)$. An examination of 5.4.5 - 5.4.7 shows that the clones and zombies of a pre-critical ghost packet reside on the same channel exactly when the original packet is sent out of the process D , in which case it generates one clone and one zombie on each relevant channel. In this case $X = \pm G$ and the labeling forces the clone to precede the zombie, and therefore in this case k_1 is the clone and k_2 is the zombie of the same original packet.

- T comprises a trigger followed by a single multicast of flush packets. This happens if T is triggered by the processing of an acknowledgement packet that cleared out **BcastWaitSet** while **FwdWaitSet** was already empty. In this case there is a single packet at each relevant channel and therefore $\text{origin}(k_1) = \text{origin}(k_2)$. An examination of 5.4.5 - 5.4.7 shows that distinct clones of a pre-critical flush packet reside on different channels, contrary to our assumption that k_1 and k_2 reside on the same channel. So this case does not occur.
- T comprises a trigger followed by a single multicast of ghost packets, followed by a single multicast of flush packets. This happens if T is triggered by the processing of an acknowledgement packet that cleared out **FwdWaitSet** while **BcastWaitSet** was already empty, or if T is triggered by the processing of a $\mathbf{n}_{\text{REM}}\langle P \rangle$ notification that occurred when **FwdQueue**[P], **FwdWaitSet** and **BcastWaitSet** were all empty. In this case there is in each relevant channel a single ghost packet followed by a single flush packet. There are two possibilities here. If T occurs in process D (in which case $X = \pm G$) then the flush packet generates no clones and the ghost packet is the original of both k_1 and k_2 which are a clone and zombie, respectively, of their shared original. If $X \neq \pm G$ then the ghost packet and the flush packet each generate a single clone (but no zombie) on each relevant packet, where the labeling forces the ghost clone to precede the flush clone. Therefore k_1 is the ghost clone and k_2 is the flush clone, and the clones are sent in the same order as their originals.
- T comprises a trigger and no other event. This happens if T is triggered by the processing of a ghost packet, or if T is triggered by a flush packet that did not cause the installation of a new view, or if T is triggered by a flush packet that did cause the installation of a new view while **LaunchQueue** was empty, or if T was triggered by the processing of an acknowledgement packet such that **FwdWaitSet** was not empty when processing started and remained non-empty when processing concluded, or if T was triggered by an acknowledgement packet for an original message packet such that **FwdWaitSet** was empty and **BcastWaitSet** was not empty when processing started, and **BcastWaitSet** remained non-empty when processing concluded, or if T is triggered by the processing of a $\mathbf{n}_{\text{REM}}\langle P \rangle$ notification that occurred when **FwdQueue**[P] was empty while **FwdWaitSet** was not empty. This case does not occur, of course, since we know that at least $\text{origin}(k_1)$ must have been queued to be sent as part of T .

□

Proof of Theorem 5. Some of the axioms are easy to check. The View Interval Axiom and the View Change Axiom are trivial. The Channel Axiom and the Packet Event Axiom are true by construction. The Notification Event Axiom and the Parent Axiom are trivial. The GMS Axiom follows directly from the construction of notifications in H^r .

The Notification Order Axiom says that view notifications are processed in order. This property is induced directly from H for all processes other than $\pm G$. In the case of $-G$ the axiom follows because we explicitly added a processing event for each notification except for the last critical one.

In the case of G , all the post-critical notification processing events (including the critical one) exist because H is transactional, while all the pre-critical ones are added explicitly. The processing order is induced directly from the labels of the events.

The Process Order Axiom has two parts. The first part claims that events within a single process are linearly ordered. The second part claims that multicast sets are finite. To prove the first part, notice that at each process P the constellation coordinate is made up of clean events at process P in H , and this set is linearly ordered since the Process Order Axiom holds in H . Within each constellation the other coordinates are also taken from linearly ordered sets: this is trivial for the adjustment and side effect coordinates since the sets \mathcal{L}_a and \mathcal{L}_f are linearly ordered. As for the sub-transaction coordinate, we have the following cases:

- Non-critical donation and co-donation constellations have the same events in H^r as they do in H , and have the same labels. Since the labels are linearly ordered in H , they are linearly ordered in H^r .
- In a critical donation constellation, the side-effect events in H^r consist of the sub-transaction events from H (the co-donation queuing event is removed) and have additional trigger events. All of these events get their sub-transaction coordinates from the space $\{k^{qu} \in \dot{\mathbb{E}}^H \mid k \in \overrightarrow{PD^H}\}$, where P is the sender of the donation packet. This space is linearly ordered by the Process Order Axiom in H .
- In a critical co-donation constellation, the side-effect events in H^r consist of all the side-effect events in H with the addition of new trigger events. All of these events get their sub-transaction coordinates from one of two sets:

$$\begin{aligned} C_1 &= \{k^{qu} \prec \ell_{v_{crit}} \in \dot{\mathbb{E}}^H \mid k \in \overrightarrow{DP^H}\} \\ C_2 &= \{k^{qu} \succ \ell_{v_{crit}} \in \dot{\mathbb{E}}^H \mid k \in \overrightarrow{GG^H}\} \end{aligned}$$

Each of these sets is linearly ordered by the Process Order Axiom in H . Taken together they are still linearly ordered because all the elements in C_1 precede $\ell_{v_{crit}}$ and all the elements in C_2 succeed it.

- In a non-donation constellation, all the events have the $\hat{0}$ value for the sub-transaction coordinate.

It follows that the λ function linearly orders $\mathbb{E}_P^{H^r}$ at each process P . By Corollary 8 the first part of the axiom now follows.

The second part of the axiom follows immediately from Lemma 22.

To verify the Process Liveness Axiom and the Piggyback Axiom, notice that they trivially hold for packets exchanged between non- G processes. As for original packets exchanged between G and non- G processes, almost nothing changes except that from the point of view of the non- G processes in H^r , process G had been a member from the start. This does not invalidate either axiom.

The clones and zombies of timely packets are either queued at $\pm G$ or dequeued at $\pm G$ at essentially the same time that the original is queued or dequeued at D (any differences in the label of the original versus the clone or zombie occur at the second or third coordinate). The same is true of untimely clones and zombies as far as queuing time is concerned. As for processing time, the first

label coordinate for the processing event of untimely clones or zombies, when such an event exists, is either $\text{Crit}(P \rightarrow G)$ or $\text{Crit}(G \rightarrow P)$ for some non- G process P . Since this label corresponds to the processing transaction of a P -donation at G or a G -codonation at P in the original history, the axioms follow in this case from the same axioms in the original history as they pertain to the donation or co-donation packet.

As for clones of a post-critical packet $k \in \overrightarrow{GG}$, such a clone is created only if the original packet is queued while P appears alive to G , and its processing event label has $\text{Crit}(G \rightarrow P)$ as a first coordinate, which verifies the axiom using the same argument as before.

The Self Channel Axiom holds trivially for original packets k . The only cloned packets on self channels exist on \overrightarrow{GG} and $\overrightarrow{-G-G}$. If such a cloned packet is pre-critical, then the original Self Channel Axiom guarantees that it is the clone of a timely packet and the axiom follows easily from that in that case. Since neither of the two channels contains clones of post-critical packets, we are done.

The Request Event Axiom is trivially induced from H .

The Order Foundation Axiom follows from Corollary 8 as long as \mathfrak{L} is itself very well founded. This in turn follows from the Order Foundation Axiom in H .

The Minimal Order Axiom is true in H^r by definition (see 5.4.9).

The First Halting Axiom would follow if we show that we only add a finite number of new events in H^r . Indeed, all the timely and untimely packets that we add are clones and zombies of packets that are queued before the critical view change notification. It follows from the Order Foundation Axiom and the View Change Axiom (in H) that there is only a finite number of such packets.

The post-critical packets that we add are all clones of post-critical packets in \overrightarrow{GG} that are queued before G processes a donation packet from some original process $P \in \mathbb{S}_{\text{vrit}}$ or before G receives a removal notification for P . If G halts then there is a finite number of such packets by the First Halting Axiom (in H). If G does not halt but P is removed then G processes the removal notification of P (because H is transactional) and therefore queues only a finite number of packets before that event by the Order Foundation Axiom.

If G does not halt and P is not removed then P does not halt (because H is conforming and not stunted, see Lemma 6). In that case P must dequeue and process to completion the join notification of G (because H is transactional) and therefore must queue a donation packet d to G . All the packets queued by P , including d , must be sent (by the Third Halting Axiom) and none of them are dropped (because there are no dropped packets in a transactional history). Therefore G receives all of them, and since G does not halt it must dequeue all of them, including d (by the Third Halting Axiom). Therefore G queues a finite number of packets prior to the $d^{\text{PR}} = \text{Crit}(P \rightarrow G)$ event (by the Order Foundation Axiom) and we are done.

The Second Halting Axiom carries over to H^r for every non- G process. The first part of the axiom holds for $\pm G$ because there are no dropped notifications in H^r . The second part holds for $-G$ because it halts for G because it processes all of its notifications.

The Third Halting Axiom has three parts. The first two parts follow from the same axiom in H and the fact that each clone and zombie k which is not processed is dropped. The third part follows for the same reason, though not so trivially. Suppose that a packet k meets the criteria of the third

part of the axiom. Then k is received and therefore is not dropped. If k is a clone or a zombie then k must be processed because all unprocessed clones and zombies are dropped. If it is original and all its predecessors are received, then surely its original predecessors are received, and therefore in H the packet k is processed, and therefore it is processed in H^r as well.

The Fourth Halting Axiom is trivially induced from H .

The main challenge is verifying the last remaining axiom, the Packet Order Axiom, which says that the channels are FIFO. This is crucial but unfortunately quite tedious. Our basic approach is the following. We take two packets k_1 and k_2 that meet the assumptions of the axiom. Namely, they belong to the same channel $\overrightarrow{XY}^{H^r}$, with $k_1^{\text{QU}} \prec k_2^{\text{QU}}$, and with k_2^{PR} existing. To verify the axiom we have to show that k_1^{PR} exists and precedes k_2^{PR} . To do that we look at the original packets $\text{origin}(k_1)$ and $\text{origin}(k_2)$ and use the FIFO property in H to glean information about their events and labels. From there we derive information about the labeling of the k_1 and k_2 events, which in turn determines their order in H^r (see Corollary 8).

We look at different cases according as k_1 and k_2 are pre-critical or post-critical. The condition $k_1^{\text{QU}} \prec k_2^{\text{QU}}$ implies that if k_2 is pre-critical, then so is k_1 . Also notice that a packet k is pre-critical if and only if $\text{origin}(k)$ is pre-critical.

Case I: k_1 and k_2 are both pre-critical

In this case we can use Lemma 23 and conclude that $\text{origin}(k_1)^{\text{QU}} \preceq \text{origin}(k_2)^{\text{QU}}$. According to the same lemma, if there is an equality then $\text{origin}(k_1) = \text{origin}(k_2)$, the common origin is a ghost packet and $X = \pm G$. In this case a routine inspection of the different cases of 5.4.5 - 5.4.7 shows that k_1^{PR} exists whenever k_2^{PR} does, and precedes the latter as well. So we can assume that $\text{origin}(k_1)^{\text{QU}} \prec \text{origin}(k_2)^{\text{QU}}$ and therefore $\text{origin}(k_1)$ and $\text{origin}(k_2)$ are distinct.

If $\text{origin}(k_2)$ is timely then the Packet Order Axiom for H implies that $\text{origin}(k_1)^{\text{PR}}$ exists and moreover $\text{origin}(k_1)^{\text{PR}} \prec \text{origin}(k_2)^{\text{PR}}$. This in turn implies that $\text{origin}(k_1)$ is a timely packet and therefore k_1 is timely as well and therefore k_1^{PR} exists. Let T_i be the transaction triggered by $\text{origin}(k_i)^{\text{PR}}$. Then $\text{trig}(T_1) \prec \text{trig}(T_2)$ and so $\ell_{T_1} \prec \ell_{T_2}$.

Going over 5.4.5 - 5.4.7 one can verify that if k is a timely clone or zombie packet then $\lambda(k^{\text{PR}})$ shares the same constellation coordinate with $\lambda(\text{origin}(k)^{\text{PR}})$. Therefore the constellation coordinate of k_i^{PR} is ℓ_{T_i} and it follows from Corollary 8 that $k_1^{\text{PR}} \prec k_2^{\text{PR}}$ and we are done.

We are left with the case where $\text{origin}(k_2)$ is untimely. We know that k_2^{PR} exists. Going over the untimely cases of 5.4.5 - 5.4.7 one can verify that $\lambda(k_2^{\text{PR}}) = [\ell_T.\text{origin}(k_2)^{\text{QU}}.*|\hat{0}]$ where $\ell_T = \text{Crit}(P \rightarrow G)$ or $\ell_T = \text{Crit}(G \rightarrow P)$. In particular we know that T exists. In this case if $\text{origin}(k_1)$ is timely then k_1^{PR} exists and is timely and therefore precedes the untimely k_2^{PR} . If $\text{origin}(k_1)$ is untimely, it follows from the existence of T that k_1^{PR} also exists and has a label $[\ell_T.\text{origin}(k_1)^{\text{QU}}.*|\hat{0}]$. This implies that $\lambda(k_1^{\text{PR}}) < \lambda(k_2^{\text{PR}})$ and therefore $k_1^{\text{PR}} \prec k_2^{\text{PR}}$.

Case II: k_1 is pre-critical and k_2 is post-critical

The case where k_1 is timely is trivial, because for a timely packet k_1^{PR} exists by definition and again by definition $k_1^{\text{PR}} \prec v_{\text{crit}}(Y)^{\text{PR}} \prec k_2^{\text{PR}}$. So we can assume that k_1 is untimely.

Since we do not add any new packets to non- G channels, and since any channel involving $-G$ does not contain any post-critical packets, we can confine our attention to the channels \overrightarrow{GP} , \overrightarrow{PG} and \overrightarrow{GG} , where P is an non- G process.

Look at the channel \overrightarrow{PG} . We do not add any new packets to this channel that are post-critical. Therefore k_2 must be original. Therefore we can assume that k_1 is not original (otherwise there is nothing to prove). We know that k_2^{PR} exists. Denote by T the transaction that is triggered by k_2^{PR} .

We know that the first post-critical packet (in H) that P queues to \overrightarrow{PG} is the donation packet d , and since P queues some post-critical packet to G in H (namely k_2), it must also queue d . Moreover, $k_2 \neq d$, since k_2 survives in H^r and d does not (see 5.4.5). Therefore $d^{\text{QU}} \prec k_2^{\text{QU}}$ and the Packet Order Axiom in H implies that d^{PR} exists and precedes k_2^{PR} in H . Therefore $\text{Crit}(P \rightarrow G)$ exists and $\text{Crit}(P \rightarrow G) \prec \ell_T$. Since k_1 is a clone or a zombie, the existence of $\text{Crit}(P \rightarrow G)$ implies that k_1^{PR} exists and $\lambda(k_1^{\text{PR}}) = [\text{Crit}(P \rightarrow G).*.|\hat{0}]$. On the other hand $\lambda(k_2^{\text{PR}}) = [\ell_T.*.*|\hat{0}]$ and $\text{Crit}(P \rightarrow G) \prec \ell_T$. Therefore $\lambda(k_1^{\text{PR}}) < \lambda(k_2^{\text{PR}})$ and therefore $k_1^{\text{PR}} \prec k_2^{\text{PR}}$ and we are done.

Look at the channel \overrightarrow{GP} . Since \overrightarrow{GP}^H does not have pre-critical packets, k_1 cannot be original. We assume first that k_2 is original. We proceed just like in the case of \overrightarrow{PG} . We know that the first post-critical packet (in H) that G queues to \overrightarrow{GP} is the co-donation packet c , and since G queues some post-critical packet to \overrightarrow{GP} in H (namely, k_2), it must also queue c . Moreover, $k_2 \neq c$, since k_2 survives in H^r and c does not (see 5.4.6). Therefore $c^{\text{QU}} \prec k_2^{\text{QU}}$ and the Packet Order Axiom in H implies that c^{PR} exists and precedes k_2^{PR} in H . Therefore $\text{Crit}(G \rightarrow P)$ exists and $\text{Crit}(G \rightarrow P) \prec \ell_T$, where T is the transaction triggered by k_2^{PR} . Since k_1 is a clone or a zombie, the existence of $\text{Crit}(G \rightarrow P)$ implies that k_1^{PR} exists and $\lambda(k_1^{\text{PR}}) = [\text{Crit}(G \rightarrow P).*.|\hat{0}]$. On the other hand $\lambda(k_2^{\text{PR}}) = [\ell_T.*.*|\hat{0}]$ and $\text{Crit}(G \rightarrow P) \prec \ell_T$. Therefore $\lambda(k_1^{\text{PR}}) < \lambda(k_2^{\text{PR}})$ and therefore $k_1^{\text{PR}} \prec k_2^{\text{PR}}$ and we are done.

If k_2 is not original then $\text{origin}(k_2) \in \overrightarrow{GG}$ (this is the only channel that produces post-critical clones). By assumption k_2^{PR} exists, so by 5.4.6 $\text{Crit}(G \rightarrow P)$ exists and

$$\lambda(k_2^{\text{PR}}) = [\text{Crit}(G \rightarrow P).\text{origin}(k_2)^{\text{QU}}.\hat{0}|\hat{0}]$$

In particular the co-donation packet $c \in \overrightarrow{GP}$ exists, and is processed, in H .

k_1 is an untimely clone or a zombie. The 5.4.6 construction shows that when $\text{Crit}(G \rightarrow P)$ exists, k_1^{PR} exists and $\lambda(k_1^{\text{PR}}) = [\text{Crit}(G \rightarrow P).\text{origin}(k_1)^{\text{QU}}.\hat{0}|\hat{0}]$. In the clean event order on \mathbb{E} we have

$$\text{origin}(k_1)^{\text{QU}} \prec \ell_{\text{vcrit}} \prec \text{origin}(k_2)^{\text{QU}}$$

Therefore $\lambda(k_1^{\text{PR}}) < \lambda(k_2^{\text{PR}})$ and therefore $k_1^{\text{PR}} \prec k_2^{\text{PR}}$ and we are done in this case as well.

We are left with channel \overrightarrow{GG} . This channel does not contain untimely packets, so there is nothing to consider in this case.

Case III: k_1 and k_2 are both post-critical

If both k_1 and k_2 are original packets there is nothing to prove. The only post-critical clones (there are no zombies) occur in \overrightarrow{GP} channels where P is a non- G process. So we can assume that $k_1, k_2 \in \overrightarrow{GP}$ and at least one of them is not original.

If k_2 is not original then it is a clone of a post-critical packet \overrightarrow{GG} with $\lambda(\text{origin}(k_2)^{\text{QU}}) = [\ell_T.*.*|\hat{0}]$. If $\text{Crit}(P \rightarrow G)$ exists then $\ell_T \prec \text{Crit}(P \rightarrow G)$. Since k_2^{PR} exists it follows by 5.4.6 that $\text{Crit}(G \rightarrow P)$

must exist and $\lambda(k_2^{\text{PR}}) = [\text{Crit}(G \rightarrow P).\text{origin}(k_2)^{\text{QU}}.\hat{0}|\hat{0}]$. This means in particular that the co-donation packet c from G to P exists, and is processed, in H . It also means, a-fortiori, that $\text{Crit}(P \rightarrow G)$ exists.

Suppose that k_1 is original. Since c is the first packet, in H , that is sent from G to P ; and since $k_1 \neq c$ (it survives in H^r and c does not), we have $c^{\text{QU}} \prec k_1^{\text{QU}}$. Therefore

$$\lambda(c^{\text{QU}}) = [\text{Crit}(P \rightarrow G).\hat{0}|\hat{0}|\text{CODONATE}] < \lambda(k_1^{\text{QU}})$$

On the other hand

$$\lambda(\text{origin}(k_2)^{\text{QU}}) = [\ell_T.*.\hat{0}|\ast] < [\text{Crit}(P \rightarrow G).\hat{0}|\hat{0}|\text{CODONATE}] = \lambda(c^{\text{QU}})$$

By 5.4.6, $\lambda(k_2^{\text{QU}}) = \lambda(\text{origin}(k_2)^{\text{QU}})$ and so $\lambda(k_2^{\text{QU}}) < \lambda(k_1^{\text{QU}})$ and therefore $k_2^{\text{QU}} \prec k_1^{\text{QU}}$ contrary to our assumption. So this case is not possible.

So we know that if k_2 is a clone then k_1 must be a clone as well, and since $\text{Crit}(G \rightarrow P)$ exists we know by 5.4.6 that k_1^{PR} exists and $\lambda(k_1^{\text{PR}}) = [\text{Crit}(G \rightarrow P).\text{origin}(k_1)^{\text{QU}}.\hat{0}|\hat{0}]$. Since $\text{origin}(k_1)$ and $\text{origin}(k_2)$ both belong to \overrightarrow{GG} and since

$$\lambda(\text{origin}(k_1)^{\text{QU}}) = \lambda(k_1^{\text{QU}}) < \lambda(k_2^{\text{QU}}) = \lambda(\text{origin}(k_2)^{\text{QU}})$$

we have $\text{origin}(k_1)^{\text{QU}} \prec \text{origin}(k_2)^{\text{QU}}$ and so by 5.3.2 $\lambda(k_1^{\text{PR}}) < \lambda(k_2^{\text{PR}})$ and so $k_1^{\text{PR}} \prec k_2^{\text{PR}}$ and we are done.

We are left with the case where k_2 is original and k_1 is not. Using the same argument that we used with an original k_1 , we can establish that $c^{\text{QU}} \prec k_2^{\text{QU}}$, where c is the co-donation packet sent from G to P in H . We know by assumption that k_2^{PR} exists. So we can conclude from the FIFO property in H that c^{PR} exists and $c^{\text{PR}} \prec k_2^{\text{PR}}$. In particular $\text{Crit}(G \rightarrow P)$ exists. Let T be the transaction triggered by k_2^{PR} . Then $\text{Crit}(G \rightarrow P) \prec \ell_T$.

Since k_1 is a clone and $\text{Crit}(G \rightarrow P)$ exists, we conclude from 5.4.6 that k_1^{PR} exists and

$$\lambda(k_1^{\text{PR}}) = [\text{Crit}(G \rightarrow P).\text{origin}(k_1)^{\text{QU}}.\hat{0}|\hat{0}] < [\ell_T.*.\hat{0}] = \lambda(k_2^{\text{PR}})$$

and therefore $k_1^{\text{PR}} \prec k_2^{\text{PR}}$. This concludes the proof. \square

Theorem 6. H^r is a conforming history.

Proof. We already know that H^r is a history, but we still have to confirm the conforming axioms, using the fact that H is a conforming history.

The Conforming Channel Axiom.

If either $P = (-G)$ or $Q = (-G)$ we are done because $-G$ halts and is removed in H^r . All the other channels in H^r are original H channels to which we added clones and zombies and from which we removed the critical donation and co-donation packets. Since the latter are finite in number, any original channel that is finite in H^r is also finite in H , and it follows from the conformity of H that either P is removed in H or Q halts in H . These properties carry over directly to H^r .

The Conforming Packet Axiom.

Suppose that this axiom is violated. Then there are processes X and Y and a packet $k \in \overrightarrow{XY}$ such that

- $v_i(Y) = \mathbf{n}_{\text{REM}}\langle X \rangle$ and $v_i(Y)^{\text{PR}}$ exists.
- k^{PR} exists and $v_i(Y)^{\text{PR}} \prec k^{\text{PR}}$

Since we do not change the non- G channels, at least one of X and Y must be equal to $\pm G$. If k is original then it does not violate the axiom. All the clones and zombies that we add to the $(\pm G)(\pm G)$ channels are timely, and we add only pre-critical packet processing events in channels where $X = (-G)$. Therefore k must be a clone or zombie on a \overrightarrow{GP} channel or a $\overrightarrow{P(\pm G)}$ channel.

In the case of a \overrightarrow{GP} channel k must be untimely, and since it is processed the constellation label of k^{PR} is $\text{Crit}(G \rightarrow P)$ and the critical co-donation packet is processed at P . It follows from the Conforming Packet Axiom in H that $\text{Crit}(G \rightarrow P) \prec_{\ell_r(G)}$ and we are done.

The case of a $\overrightarrow{P(\pm G)}$ channel and an untimely k is similar. If k is timely then the constellation label of k^{PR} is equal to the clean event origin(k)^{PR}. By the Conforming Packet Axiom in H we know that origin(k)^{PR} $\prec_{\ell_r(P)}$ and we are done in this case as well.

The Conforming Notification Axiom.

By construction H^r has no dropped notifications, so the axiom is vacuously true.

The Conforming GMS Axiom.

The new process $-G$ has a finite view interval. If $P \neq (-G)$ and P halts then P exists in H and has a finite view interval there. This property carries over to H^r .

The Conforming Parent Axiom.

The history H is transactional and therefore has no uninitialized processes. This property carries over to H^r for all the processes with the exception of $-G$. But $-G$ itself is also initialized because it processes all of its notifications. Therefore this property holds vacuously at H^r .

The Conforming Halt Axiom.

This property carries over to H^r for all the processes in H . The property holds for $-G$ as well because it halts.

□

6 The History Equivalence Theorem

6.1 Introduction

We are now ready to prove the fundamental property of H^r , namely, that it performs the same calculation as H .

The proof proceeds by induction on the partially ordered constellations of \mathfrak{L} . We will formulate an inductive hypothesis that correlates the state of the processes in H and in H^r prior to a given

constellation. We will show that under the hypothesis, if the processes of H^r process the triggers of the constellation according to the CBCAST algorithm, each would generate and queue the exact same packets, and in the same order, that are observed in the H^r transaction for that trigger. Moreover, the post-processing state of the processes in H^r will continue to relate to the state of the processes in H according to the inductive hypothesis once the whole constellation is processed.

The most important part of the rather elaborate inductive hypothesis is what it says about the eventual state of H and H^r , namely that the state becomes identical. This means that the calculation carried out by the two histories is the same calculation. This makes it possible to carry over desirable properties like coherence and progress from H^r to H . By repeating that step we can ultimately carry over these properties from relatively simple histories that do not have any process joins to the more intractable histories that have any finite number of such joins.

Theorem 7 (History Equivalence Theorem). *H^r is the history of a CBCAST and APP computation that performs the same computation that H does. Specifically*

- H^r delivers the same messages and view installations in the same order and at the same constellation as H does at any process $P \neq \pm G$.
- H^r delivers the same messages and view installations in the same order and at the same constellation as H does at process G after the critical moment.
- H^r delivers the same messages and view installations at processes $\pm G$ in the same order and at the same constellation as H does at process D before the critical moment.
- At some point the states of H^r and H become identical

Where a message delivery is an invocation of the ApplyMessage up-call and a view installation is an invocation of the ApplyJoin or ApplyRemoval up-calls.

6.2 Proof plan and preliminaries

The claim of the theorem is a little subtle. H is a history that arises naturally from an execution of CBCAST and APP. But H^r is a synthetic history whose trigger events occur for no underlying reason. To prove the theorem we have to endow H^r with an execution that gives rise to its arbitrary behavior. We do that inductively, one constellation at a time.

At the beginning of time we have each original process in H^r initialized by invoking the protStart call, using

$$\text{roster}^r = \text{roster} \bigcup \{\pm G\}$$

as the roster. This causes the processes of H^r to initialize to a specific initial state. We will show that this state is *similar* to the initial state of H , where similarity of state is a rather complex relationship that we will define later. This similarity forms the basis of our inductive process. The induction is by the partially ordered constellations of \mathcal{L}_c (see Definition 21).

Recall from 5.3 that H and H^r constellations were defined as sets of events that share the first coordinate, called the constellation coordinate, in their labels. Since each constellation coordinate is a clean H -trigger, there is a very close relationship between H^r -constellations and H -constellations. In fact in most cases a H^r -constellation is essentially an original H -transaction or a set of clones of an H -transaction. The exceptions are the critical donation and co-donation transactions of

H . Each of these gets broken down into a sequence of H^r transactions which correspond to H sub-transactions and which can be distinguished by their sub-transaction labels.

For each constellation we assume the following about H^r and H :

1. The starting states of H^r and H are similar.
2. The starting states of the APP thread in H^r and H are *identical*.
3. The next execution interval of the APP thread will occur at exactly the same time in H^r and H at every process. In particular, since the constellation consists of at most a single transaction per process in H , the APP thread will not continue running in H^r until the conclusion of the constellation, despite the fact that it may consist of multiple transactions.

The last assumption reflects a degree of freedom that we have in weaving the APP computation into H^r . After all, we only have to show that H^r *can* arise as a history of a CBCAST and APP computation, not that it must arise.

The only difficulty with the timing of the APP thread occurs at its inception. In H , the APP thread at G is launched when G installs the critical view. In H^r however the thread is launched at the beginning of time by the protStart procedure, since G is original in H^r . However the launch is asynchronous, meaning that the thread is not executed immediately but at some indeterminate point in the future. Since the launch is earlier in H^r we can simply assume that the execution is delayed long enough to coincide with the execution in H . At process $-G$ in H^r the launch also occurs at the beginning of time but we can assume that the execution is delayed until $-G$ halts and therefore never occurs.

Under these assumptions we demonstrate the following conclusions:

- Using the CBCAST callbacks to execute the constellation in H^r results in an ending state of H^r that is similar to the state of H at the end of the same constellation.
- The side effects that are generated by the CBCAST callbacks are identical to the observed side effects in H^r . In other words the current constellation looks like it has come about as a result of a CBCAST execution rather than an arbitrary choice of side effects.
- The message deliveries and view installations that are generated by the CBCAST callbacks in H^r are identical to those generated in H , in the sense that was elaborated in the statement of the theorem. As a result any information that is visible to APP remains identical at the end of the constellation.

This looks like it is enough for carrying the induction forward, but it is not quite enough, because even though we have shown that the side effects of the current constellation are generated by CBCAST rather than being arbitrary, we have not shown that any subsequent trigger events are non-arbitrary.

So suppose that C is a constellation in H^r , and suppose that every preceding constellation has been shown to arise from a CBCAST and APP execution. Why should this execution give rise to the triggers of C ? Look at any trigger in C :

If the trigger is a dequeuing of a notification then there is no problem, because GMS is part of H^r and we are allowed to control its behavior arbitrarily.

If the trigger is a dequeuing event of a packet, then the packet was queued in a previous constellation and therefore was a result of a **CBCAST** and **APP** execution. The dequeuing of the packet at this point is the result of the timing (or labeling) that we built into H^r explicitly in order to have packets dequeue at their destinations at improbable, but incredibly convenient times.

If the trigger is a dequeuing of a message broadcast request then its existence depends on **APP** actually having produced the same requests in H^r and in H at the same time. The only way for this to happen is for the **APP** thread to be exactly identical in both histories, and this can only be guaranteed by perfectly masking the differences between H^r and H from **APP**. For this to happen we need three conditions:

- The **APP** thread must have been in an identical state in both histories at all processes at the end of the previous constellation.
- The **APP** thread must have seen the exact same information since that time.
- The **APP** thread execution must have proceeded at the exact same speed at the exact same intervals in both histories and must not have intermingled with constellation executions.

We have shown that the first two conditions are met, and are at liberty to assume the third, as we have seen. Therefore we can conclude that **APP** could have issued the same requests³

With these observations we can conclude that the triggers of C are indeed a result of a **CBCAST** and **APP** execution and that the three inductive hypotheses (1), (2) and (3) above continue to hold.

The definition of state similarity is somewhat complex and varies depending on the *period* in the life of the process into which the constellation falls. Each process goes through three periods, the *pre-critical*, *interim* and *convergent* periods. The pre-critical period includes all the transactions that occur prior to the critical notification and the convergent period includes all the transactions that occur after the critical view is installed. The interim period includes all the constellations that occur while the critical view is pending installation.

Here is the crux of the matter: in the convergent period, similarity becomes equality, and the two histories converge as claimed.

It turns out that desirable properties like Causal Order and Progress carry over from H^r to H . By iterating the history reduction process we can ultimately carry over these properties from relatively simple histories that do not have any process joins to the more intractable histories that have any finite number of such joins. Later we will show that join-free histories enjoy the Causal Order Property and the Progress Property. As a result both properties hold for finite-join histories. We will show that the Causal Order property holds for histories with an infinite number of joins as well. This is not true for the Progress Property.

6.3 Side Effects in H^r

In our model each side effect is a queuing event. A queuing event is a multicast (in the case of a message, ghost or flush packet) or a unicast (in the case of an acknowledgement, donation or

³we could have simplified this argument by replacing **APP** in H^r with a random oracle that by sheer luck broadcasts the same messages that **APP** issues. However we thought it was significant that the argument could be carried forward with the same user application and without resorting to an artificial oracle.

co-donation packet). In a queuing event a process P queues a set of identical packets to outbound channels, bound for a target set of processes (see the Packet Event Axiom). Before we can make inductive arguments, we must relate the observed side effects in H^r to the observed side effects in H .

As we prove the History Equivalence Theorem, we will repeatedly make the argument that the execution of **CBCAST** in H^r produces the observed H^r side effects. Each observed side effect in H^r has two characteristics: the type and content of the packet that is being queued, and the target set of the multicast or unicast. In each case we will have to show that the **CBCAST** code execution produces the observed type of packet with the observed content. As for the target set, all the multicasts in **CBCAST** (step 2 of **protBroadcast**, step 5 of **protRemove**, steps 1 and 2 of **CheckFlush**) use **ContactSet** as the target set. We will establish later (see Lemma 24) that in each case, the observed target set is equal to the value of **ContactSet** that exists at the process in H^r at the time of the multicast. Unicasts will not present a similar problem.

6.4 The inductive hypothesis

The inductive hypothesis relates the states of certain processes in H and H^r . The complete set of variables that make up the state of a process is listed in 3.4.1. The inductive hypothesis is complex enough to warrant a preliminary discussion.

Thanks to labeling we have a common "timeline" for H and H^r , namely the common constellation partial order. We divide this common timeline into three periods: The pre-critical period, the interim period and the convergence period. The pre-critical period is the interval of time up to the critical view change constellation ℓ_{vcrit} . The interim period ends at a process when that process installs the critical view. This boundary occurs at a different constellation at each process. Process -G is removed at the end of the pre-critical period, so it does not have an interim period. The convergence period starts at the end of the interim period and continues indefinitely. The inductive hypothesis is divided into separate hypotheses for each time period. The most important of those is the convergent period, where the hypothesis is that H and H^r are identical.

To summarize, let $e \in \dot{\mathbb{E}}$ be a constellation and let P be a process. Then

- the constellation belongs to the *pre-critical period* at P if $e \prec \ell_{\text{vcrit}}$
- the constellation belongs to the *interim period* at P if $e \succeq \ell_{\text{vcrit}}$ and $\text{cur_view}_{P@e} < v_{\text{crit}}$
- the constellation belongs to the *convergent period* at P if $e \succeq \ell_{\text{vcrit}}$ and $\text{cur_view}_{P@e} \geq v_{\text{crit}}$

The most complex constellations are the critical donation and co-donation constellations, $\text{Crit}(P \rightarrow G)$ and $\text{Crit}(G \rightarrow P)$. Each of these is a single transaction in H , but becomes a sequence of transactions in H^r - potentially even an empty sequence, in which case the constellation does not exist in H^r . To prove the inductive hypothesis for one of these constellations, we need to resort to a second level of induction. For this purpose we will formulate sub-hypotheses that relate the states of H and H^r at each sub-transaction.

We build on our observations in 6.3 to define the following equivalence between side effects. We use this equivalence to separate the issue of side effect type and content from the issue target set, that will be treated separately.

Definition 26. Let $e = k^{\text{qu}}$ and $e^r = k^{r\text{qu}}$ be two queuing events in H and H^r , respectively. We say that e and e^r produce an **equivalent** side effect if k and k^r have the same packet type and $\text{cont}(k) \cong \text{cont}(k^r)$ (see Definition 22).

The inductive hypothesis is actually a set of related hypotheses and sub-hypotheses. The main hypotheses are:

- The First Pre-Critical Hypothesis, which describes how the state of a process $P \in \mathbb{P}^H$ in H is related to the state of the same process in H^r at the start of a pre-critical constellation. Notice that $P \neq \pm G$ because $-G$ is not a process in H while G joins post-critically in H .
- The Second Pre-Critical Hypothesis, which describes how the states of D , G and $-G$ in H^r are related to each other at the start of a pre-critical constellation. Notice that in this case the comparison is within H^r , not between H and H^r .
- The Interim Non- G Hypothesis, which describes how the state of a process $P \neq G$ in H is related to the state of the same process in H^r at the start of a post-critical constellation that occurs before P installs the critical view in H .
- The Interim G Hypothesis, which describes how the state of G in H is related to its state in H^r at the start of a post-critical constellation that occurs before G installs the critical view in H .
- The Convergent Hypothesis, which claims that the state of a process P in H is identical to its state in H^r at any time after P installs the critical view in H .

The sub-hypotheses are:

- The Donation Sub-Hypothesis, which describes how the state of G in H relates to its state in H^r at the start of each sub-transaction of each donation constellation at G .
- The First Co-Donation Sub-Hypothesis, which describes how the state of $P \neq G$ in H relates to its state in H^r at the start of each untimely sub-transaction of each co-donation constellation in P .
- The Second Co-Donation Sub-Hypothesis, which describes how the state of $P \neq G$ in H relates to its state in H^r at the start of each post-critical sub-transaction of each co-donation constellation in P .

Inductive Hypothesis 1 (First Pre-Critical Hypothesis). *Let C be any pre-critical constellation and let P be a process that exists in both H and H^r at the start of C . Then the state of P in H*

and the state of P in H^r are identical at that point, with the following exceptions:

$$\begin{aligned}
MSet^r &= MSet \bigcup \{\pm G\} \\
LiveSet^r &= LiveSet \bigcup \{\pm G\} \\
ContactSet^r &= ContactSet \bigcup \{\pm G\} \\
vt^r[] &= vt[] \bigcup \{[G] = 0, [-G] = 0\} \\
ReceiveSet^r &\cong ReceiveSet \\
FwdQueue^r[] &\cong FwdQueue[] \bigcup \{[G] = \emptyset, [-G] = \emptyset\} \\
WaitSet^r &\cong \{\langle msg, index, \partial(iset) \rangle \mid \langle msg, index, iset \rangle \in WaitSet\} \\
\text{where } \partial(iset) &= \begin{cases} iset & D \notin iset \\ iset \bigcup \{iset[\pm G] = \{f = iset[D].f, b = 0\}\} & D \in iset \end{cases} \\
mpkt_in^r[X].f &= \begin{cases} mpkt_in[X].f & X \neq \pm G \\ mpkt_in[D].f & X = \pm G \end{cases} \\
mpkt_in^r[X].b &= \begin{cases} mpkt_in[X].b & X \neq \pm G \\ 0 & X = \pm G \end{cases} \\
ghost^r[] &= ghost[] \bigcup \{[G] = ghost[D], [-G] = ghost[D]\} \\
flush^r[] &= flush[] \bigcup \{[G] = ghost[D], [-G] = ghost[D]\}
\end{aligned}$$

Notice that $flush^r[\pm G]$ is indeed inherited from $ghost[D]$, not $flush[D]$!

Inductive Hypothesis 2 (Second Pre-Critical Hypothesis). *Let C be any pre-critical constellation. Then the states of $\pm G$ in H^r at the start of C are identical to the state of D in H^r at the same point, with the following exceptions:*

$$\begin{aligned}
self^r(\pm G) &= \pm G \\
BcastWaitSet^r(\pm G) &= \emptyset \\
FwdWaitSet^r(\pm G) &= \{\langle msg, \{f = index.f, b = 0\}, iset \rangle \mid \\
&\quad \langle msg, index, iset \rangle \in FwdWaitSet^r(D) \} \\
LaunchQueue^r(\pm G) &= \emptyset \\
flush_height^r(\pm G) &= ghost_height^r(D) \\
mpkt_out^r.b(\pm G) &= 0
\end{aligned}$$

Inductive Hypothesis 3 (Interim Non- G Hypothesis). *Let C be any post-critical constellation and let $P \neq G$ be a process that exists at the start of C and has no yet installed the critical view in H . Then the state of P in H is identical to the state of P in H^r at that point with the following*

exceptions:

$$\begin{aligned}
MSet^r &= MSet \bigcup \{\pm G\} \\
PendViewQueue^r &= PendViewQueue \text{ with } \langle \text{REMOVE}, -G \rangle \text{ replacing } \langle \text{JOIN}, G \rangle \\
vt^r[] &= vt[] \bigcup \{[G] = 0, [-G] = 0\} \\
ReceiveSet^r &\cong ReceiveSet \\
FwdQueue^r[] &\cong FwdQueue[] \\
WaitSet^r &\cong WaitSet
\end{aligned}$$

Inductive Hypothesis 4 (Interim G Hypothesis). *Let C be any post-critical constellation and suppose that G exists at the start of C and has not yet installed the critical view in H . Then the state of G in H is identical to the state of G in H^r at that point with the following exceptions:*

$$\begin{aligned}
MSet^r &= MSet \bigcup \{\pm G\} \\
PendViewQueue^r &= PendViewQueue \text{ with } \langle \text{REMOVE}, -G \rangle \text{ replacing } \langle \text{JOIN}, G \rangle \\
ContactSet^r &= LiveSet \\
vt^r[] &= vt[] \bigcup \{[G] = 0, [-G] = 0\} \\
ReceiveSet^r &\cong ReceiveSet \\
FwdQueue^r[] &\cong FwdQueue[] \\
WaitSet^r &\cong WaitSet
\end{aligned}$$

Inductive Hypothesis 5 (Donation Sub-Hypothesis). *Let P be any process that sends a critical donation packet to G in H and let k be any untimely packet in \overrightarrow{PD} . If G processes the critical donation packet from P in H then the state of G in H at the start of the $[\text{Crit}(P \rightarrow G).k^{\text{qu}}.*|\hat{0}]$ sub-transaction is identical to the state of G in H^r at the start of the matching transaction in H^r , with the following exceptions:*

$$\begin{aligned}
MSet^r &= MSet \bigcup \{\pm G\} \\
PendViewQueue^r &= PendViewQueue \text{ with } \langle \text{REMOVE}, -G \rangle \text{ replacing } \langle \text{JOIN}, G \rangle \\
ContactSet^r &= LiveSet \\
vt^r[] &= vt[] \bigcup \{[G] = 0, [-G] = 0\} \\
ReceiveSet^r &\cong ReceiveSet \\
FwdQueue^r[] &\cong FwdQueue[] \\
WaitSet^r &\cong WaitSet \\
ghost[P] &\leq ghost^r[P] \leq ghost_height_{P@l_{\text{vcrit}}} \\
flush[P] &\leq flush^r[P] \leq flush_height_{P@l_{\text{vcrit}}}
\end{aligned}$$

Inductive Hypothesis 6 (First Co-Donation Sub-Hypothesis). *Let $P \neq G$ be any process that receives a critical co-donation packet from G in H and let k be any untimely packet in \overrightarrow{DP} . If P processes the critical co-donation in H then the state of P in H at the start of the $[\text{Crit}(G \rightarrow P).k^{\text{qu}}.*|\hat{0}]$*

sub-transaction in H is identical to the state of P in H^r at the start of the matching transaction in H^r , with the following exceptions:

$$\begin{aligned}
MSet^r &= MSet \bigcup \{\pm G\} \\
PendViewQueue^r &= PendViewQueue \text{ with } \langle \text{REMOVE}, -G \rangle \text{ replacing } \langle \text{JOIN}, G \rangle \\
vt^r[] &= vt[] \bigcup \{[G] = 0, [-G] = 0\} \\
ReceiveSet^r &\cong ReceiveSet \\
FwdQueue^r[] &\cong FwdQueue[] \\
WaitSet^r &\cong WaitSet \\
ghost[G] &\leq ghost^r[G] \leq ghost_height_{D@l_{v_{crit}}} \\
flush[G] &\leq flush^r[G] \leq ghost_height_{D@l_{v_{crit}}}
\end{aligned}$$

Inductive Hypothesis 7 (Second Co-Donation Sub-Hypothesis). *Let $P \neq G$ be any process that receives a critical co-donation packet from G in H and let k be any uncontacted packet in \overrightarrow{GG} . If P processes the critical co-donation in H then the state of P in H at the start of the $[Crit(G \rightarrow P).k^{qu}.*|\hat{0}]$ sub-transaction is identical to the state of P in H^r at the start of the matching transaction in H^r , with the following exceptions:*

$$\begin{aligned}
MSet^r &= MSet \bigcup \{\pm G\} \\
PendViewQueue^r &= PendViewQueue \text{ with } \langle \text{REMOVE}, -G \rangle \text{ replacing } \langle \text{JOIN}, G \rangle \\
vt^r[] &= vt[] \bigcup \{[G] = 0, [-G] = 0\} \\
ReceiveSet^r &\cong ReceiveSet \\
FwdQueue^r[] &\cong FwdQueue[] \\
WaitSet^r &\cong WaitSet \\
ghost[G] &\leq ghost_height_{D@l_{v_{crit}}} \leq ghost^r[G] \leq ghost_height_{G@Crit(P \rightarrow G)} \\
flush[G] &\leq ghost_height_{D@l_{v_{crit}}} \leq flush^r[G] \leq ghost_height_{G@Crit(P \rightarrow G)}
\end{aligned}$$

Inductive Hypothesis 8 (Convergent Hypothesis). *For any post-critical constellation C and any process P that exists at the start of C , if P has already installed the critical view at that point in H , then the state of P in H is identical to the state of P in H^r at that same point. Moreover, the state of P does not contain any pre-critical messages.*

6.5 Technical lemmas for the History Equivalence Theorem proof

Lemma 24. *Let $e = k^{qu}$ be a queuing event of a ghost, flush or message packet at process P in H^r , and let $e_0 = \text{origin}(k)^{qu}$ be the original queuing event at process R in H . Assume that the inductive hypothesis holds and that one of the following conditions hold as well:*

1. e is pre-critical, and $ContactSet^r_{P@e} = ContactSet_{R@e_0} \bigcup \{\pm G\}$
2. e is post-critical, $P \neq G$ and $ContactSet^r_{P@e} = ContactSet_{R@e_0}$

3. e is post-critical, $P = G$ and $\text{ContactSet}^r_{P@e} = \text{LiveSet}_{R@e_0}$

Then $T_e = \text{ContactSet}^r$

Proof. Most of the cases follow immediately from Lemma 22 and the various inductive hypotheses. The one non-trivial case is when e is post-critical, $P = G$ and the Convergent Hypothesis holds. In that case it follows from Lemma 22 and the inductive hypothesis that $T_e = \text{LiveSet}^r$. We have to show that $\text{ContactSet}^r = \text{LiveSet}^r$. It follows from Lemma 8(2) that the equality holds for original processes. Since G is an original process in H^r we are done. \square

Lemma 25. *Let msg be an unstamped message and let $P \neq \pm G$ be a process in H . Suppose that the states of P in H and H^r satisfy the First Pre-Critical Hypothesis. Suppose as well that $D \in \text{LiveSet}$ and $\pm G \notin \text{LiveSet}$ at P in H . Then executing the $\text{protBroadcast}(\text{msg})$ in both histories will result in equivalent side effects (see Definition 26) while preserving the Hypothesis.*

Proof. We follow the execution of the protBroadcast procedure step by step.

The first step is a decision whether to proceed or to queue msg to LaunchQueue . Both histories take the same decision here, resulting in identical changes to LaunchQueue . If $v_gap > 0$ we are done.

The next step increments $\text{mpkt_out}.b$, keeping it identical.

The next two steps define a temporary vector vt' . Since $P \neq \pm G$, the resulting vector has values at H and H^r that bear the same relationship to each other as vt does.

The next three steps stamp the message. By Definition 22 this produces equivalent messages.

The next step creates the queuing event. By Definition 26 this event produces equivalent side effects.

The next step creates the local variable index which has the same value in both histories.

The next step creates the local vector $\text{iset}[]$, which is related in H and H^r the same way $\text{mpkt_in}[]$ is.

Since $D \in \text{LiveSet}$, we know from Lemma 8 that there is a D coordinate in $\text{mpkt_in}[]$ and therefore the record that is added to BcastWaitSet in the last step bears the required relation for WaitSet records. Therefore this step preserves the inductive hypothesis and we are done. \square

Lemma 26. *Let P and Q be two processes that use the same implementation of the ApplyMessage up-call. Further assume that the states of P and Q have the following relations:*

$$\begin{aligned} \text{ReplicatedData}(P) &= \text{ReplicatedData}(Q) \\ \text{cur_view}(P) &= \text{cur_view}(Q) < v_{\text{crit}} \\ \text{MSet}(P) &= \text{MSet}(Q) \bigcup \{\pm G\} \\ \text{ReceiveSet}(P) &\cong \text{ReceiveSet}(Q) \\ vt[] (P) &= vt[] (Q) \bigcup \{[G] = 0, [-G] = 0\} \end{aligned}$$

Then executing the *Scan* procedure in both processes will result in the same relations being preserved, with all other state variables remaining unchanged in both P and Q . Moreover both processes invoke the *ApplyMessage* up-call at the same times and with the same messages.

Proof. We follow the execution of the *Scan* procedure step by step.

The first step sets the value of *deliverable_messages_found* to *false* at both P and Q .

The next step is a loop over members of *ReceiveSet*. Since *ReceiveSet* is equivalent at P and Q , the loop goes over the same messages in the same order in both executions. For each message, we check whether the message is a current view message and whether we have already delivered all the previous messages from the same source. This amounts to checking whether $\text{VIEW}(\text{msg}) = \text{cur_view}$ and whether $\text{VT}(\text{msg})[\text{ORIG}(\text{msg})] = \text{vt}[\text{ORIG}(\text{msg})] + 1$.

The first check gives the same result in P and Q because $\text{VIEW}(\text{msg})$ is the same (due to equivalence) and *cur_view* is assumed to be the same. Moreover, if the check is successful it means that $\text{VIEW}(\text{msg}) = \text{cur_view} < v_{\text{crit}}$ and therefore $\text{ORIG}(\text{msg}) \neq \pm G$. This is because G starts life with $\text{cur_view} + v_{\text{gap}} = v_{\text{crit}}$ so it does not originate any messages before $\text{cur_view} = v_{\text{crit}}$ and $v_{\text{gap}} = 0$. $-G$ does not originate any messages at all.

The second check gives the same result because of equivalence and because of our assumption that $\text{ORIG}(\text{msg}) \neq \pm G$. Therefore $\text{VT}(\text{msg})[\text{ORIG}(\text{msg})]$ and $\text{vt}[\text{ORIG}(\text{msg})]$ are the same at P and Q .

Therefore the conditional block is executed for the same messages, following the steps:

- *all_dependents_delivered* is set to *true* in both P and Q .
- A loop searches *MSet* for coordinates whose values will prevent delivery. Other than $\pm G$ the set *MSet* in Q contains the same processes as in P and the test yields the same results in both. The only way this loop could produce divergent results is if Q decided that the message was deliverable while P found an impediment that is related to $\text{pid} = \pm G$. But the equivalence condition on *ReceiveSet* guarantees that $\text{VT}(\text{msg})[\pm G] = 0$ in P , so this cannot happen and the loop must exit with the same value of *all_dependents_delivered* in both executions.
- The decision to deliver the message is controlled by *all_dependents_delivered* so both P and Q make the same decision. The message delivery takes the following steps:
 - The value of *deliverable_messages_found* is set to *true* at both P and Q .
 - The vector time is incremented at the $\text{ORIG}(\text{msg})$ coordinate. This is the same coordinate in P and Q due to equivalence, and since it is not $\pm G$, we increment an identical value in *vt*, keeping it identical.
 - The message is removed from *ReceiveSet* and stripped of its vector time and view stamps, leaving it with only its origin stamp. Since the origin stamp is identical for equivalent messages, this leaves the message identical in P and Q , in addition to leaving *ReceiveSet* equivalent.
 - The identical message is applied to the identical user data object *ReplicatedData* in the same way. This leaves the data object identical.

The last step is a recursive call to *Scan*, controlled by *deliverable_messages_found*. We have shown that up to this point the assumed relationships have not changed and no variables changed other

than some of the ones mentioned. Since P and Q take the same decision about making the recursive call, the lemma is now proven by induction. \square

Lemma 27. *Let P and Q be two processes whose states have the following relations:*

$$\begin{aligned} \text{FwdWaitSet}(P) &= \emptyset \text{ iff } \text{FwdWaitSet}(Q) = \emptyset \\ \text{BcastWaitSet}(P) &= \emptyset \text{ iff } \text{BcastWaitSet}(Q) = \emptyset \\ \text{cur_view}(P) &= \text{cur_view}(Q) \\ \text{v_gap}(P) &= \text{v_gap}(Q) \\ \text{ghost_height}(P) &= \text{ghost_height}(Q) \\ \text{flush_height}(P) &= \text{flush_height}(Q) \end{aligned}$$

Then executing the CheckFlush procedure in both processes has the following two results:

- *The same relations are preserved, with all other state variables remaining unchanged in both P and Q .*
- *The same side effects occur in P and Q .*

Proof. We follow the execution of CheckFlush step by step.

The first step exits if FwdWaitSet is not empty. P and Q make the same decision here by assumption.

The next block compares ghost_height to $\text{cur_view} + \text{v_gap}$. If ghost_height is low it updates it and broadcasts ghost packets of height $\text{cur_view} + \text{v_gap}$. Since ghost_height , cur_view and v_gap are all the same in P and Q , this block results in the same side effects and preserves the postulated relations without changing any other variables.

The rest of the procedure is a repeat of the first part with BcastWaitSet and flush_height replacing FwdWaitSet and ghost_height , so the same argument holds. \square

Lemma 28. *Let P and Q be two processes whose states have the following relations:*

$$\begin{aligned} \text{FwdWaitSet}(P) &= \emptyset \text{ iff } \text{FwdWaitSet}(Q) = \emptyset \\ \text{BcastWaitSet}(P) &= \emptyset \\ \text{cur_view}(P) &= \text{cur_view}(Q) \\ \text{v_gap}(P) &= \text{v_gap}(Q) \\ \text{ghost_height}(P) &= \text{ghost_height}(Q) \\ \text{flush_height}(P) &= \text{ghost_height}(Q) \end{aligned}$$

Then executing the CheckFlush procedure in both processes has the following two results:

- *The same relations are preserved, with all other state variables remaining unchanged in both P and Q .*
- *If Q broadcasts ghost packets of height v then P broadcasts ghost packets of height v followed by flush packets of height v .*

- If Q does not broadcast ghost packets, then P has no side effects.

Proof. We follow the execution of CheckFlush step by step.

The first step exits if FwdWaitSet is not empty. P and Q make the same decision here by assumption. If they exit then neither has side effects and we are done.

The next block compares *ghost_height* to *cur_view* + *v_gap*. If *ghost_height* is low it updates it and broadcasts ghost packets of height *cur_view* + *v_gap*. Since *ghost_height*, *cur_view* and *v_gap* are all the same in P and Q , this block results in the same side effects and preserves the postulated relations without changing any other variables.

The rest of the procedure is a repeat of the first part with BcastWaitSet and *flush_height* replacing FwdWaitSet and *ghost_height*. If P and Q decided to broadcast ghost packets, then our assumptions will force P to broadcast flush packets as well, as the lemma claims. Since the P decision to broadcast flush packets follows Q 's decision to broadcast ghost packets, the value of *flush_height* in P ends up being equal to the value final value of *ghost_height* in Q , as claimed. \square

Corollary 9. 1. Let P be a process in H and suppose that the state relationships in one of the inductive hypotheses or sub-hypotheses, excluding the Second Pre-Critical Hypothesis, hold with respect to states of P in H and H^r . Then executing the CheckFlush procedure at P in H and H^r preserves the same relations and causes the same side effects.

2. Suppose that the state relationships in the Second Pre-Critical Hypothesis hold at D , G and $-G$ in H^r . Then executing the CheckFlush procedure in all three processes preserves the same relations and causes side effects that are related as stated in Lemma 28.

Proof. \square

Lemma 29. Let X be a process in H that is in the middle of the execution of a constellation in both H and H^r (if $X = G$ then the constellation must be post-critical). Suppose that the First Pre-Critical, Interim non- G , Interim G or Convergent Hypothesis holds with respect to the state of X in H and H^r (if $X = G$ then the pre-critical case does not apply). Also assume that *ul_view* = *cur_view* ^{r} at X in H^r . Then executing the TryToInstall procedure in both histories at X has the following results:

- If *cur_view* < v_{crit} in H at the end of the execution then the same inductive hypothesis (First Pre-Critical or Interim) still holds.
- If *cur_view* $\geq v_{\text{crit}}$ in H at the end of the execution then the Convergent Hypothesis holds.
- If $P = G$ and the procedure installs the critical view then G launches the APP thread at exactly the same moment in H and H^r .
- In all cases the side effects in H^r are equivalent to the side effects in H .

Proof. We follow the execution step by step:

The procedure starts with a loop that goes over the live processes, looking for an impediment to installation of the next view in the form of *flush*[] values that are too low. In all the situations under

consideration we have

$$\begin{aligned}
cur_view^r &= cur_view \\
v_gap^r &= v_gap \\
LiveSet^r &\supset LiveSet \\
flush^r[Y] &= flush[Y] \quad \text{whenever } Y \in LiveSet
\end{aligned}$$

Therefore if there is an impediment to installation in H , the same impediment exists in H^r . The converse is trivially true in all but the pre-critical case, because that is the only case where $LiveSet^r \neq LiveSet$. However in that case we have, by the First Pre-Critical Hypothesis:

$$flush^r[\pm G] = ghost[D] \geq flush[D]$$

where the inequality on the right follows from Lemma 8(5). Therefore if G or $-G$ form an impediment in H^r , then so does D in H . Therefore both histories make the same decision on whether to proceed with installing all the pending views.

The next part of the procedure is a loop that installs all the pending views. It goes through the following steps:

- The obsolete messages are removed from `ReceiveSet`. Because $ReceiveSet^r \cong ReceiveSet$, they have the same messages and each message has the same message view. Therefore the same messages are discarded in H and H^r and the hypothesis is preserved. In the case where the view being installed is the critical one, this step leaves `ReceiveSet` with no messages of pre-critical view. By the definition of equivalence (Definition 22) this means that $ReceiveSet^r = ReceiveSet$.
- The obsolete messages are removed from `FwdQueue[]`. This is very similar to the previous step. The only complication is that in the pre-critical case there are two queues, $FwdQueue^r[\pm G]$, that do not exist in H . However these queues are empty according to the First Pre-Critical Hypothesis, and so this step preserves the inductive hypothesis as well. As in the previous case, if the view being installed is the critical one, then at this stage $FwdQueue^r[X] = FwdQueue[X]$ at every process X .
- The next two steps increment cur_view and decrement v_gap . This obviously preserves the inductive hypothesis. Also $ul_view = cur_view^r - 1$. This will be fixed soon.
- The next step pops the head of `PendViewQueue`. In most cases this trivially preserves the inductive hypothesis, as well as yielding an identical value for *notification*. However in the case where the view being installed is the critical one, this step renders `PendViewQueue` identical in both histories while yielding different values for *notification*. In H we have a value that indicates that G is joining while in H^r we have a value that indicates that $-G$ is leaving.
- The next step updates `MSet` according to the type of notification, and applies the change to the replicated application data. We have to consider the following cases:
 1. The notification is for a non-critical joining of a process which is not the local process X . Since the first join in H occurs at the critical view we have

$$cur_view \geq v_{crit} > MAGIC_VIEW$$

Both histories add the new process to **MSet**, preserving the inductive hypothesis. Then in H the process X invokes the up-call **ApplyJoin@X**, which modifies **ReplicatedData** in an unspecified user-defined way. In H^r the process X invokes **ApplyJoin^r@X**. According to 5.4.10, **ApplyJoin^r@X** behaves the same way in this case, regardless of whether $X = G$ or not. It increments ul_view and restores the equality $ul_view = cur_view^r$. Since ul_view is positive, the up-call invokes the original **ApplyJoin@X**. Therefore **ReplicatedData** is modified in the same way in H^r and the inductive hypothesis is preserved.

2. The notification is for a non-critical joining of the local process X . This case is just like the previous one but in addition process X also launches the main **APP** thread, with the same identity parameter **pid** in both histories. We may assume that the thread will start executing at the exact same time in both histories (because it is possible, not because it is probable).
3. The notification is for a non-critical removal of a process. In both histories X removes the process from **MSet**, preserving the inductive hypothesis. Then in H it invokes the up-call **ApplyRemoval@X**, which modifies **ReplicatedData** in an unspecified user-defined way. In H^r it invokes **ApplyRemoval^r@X**. If $X \neq G$ then according to 5.4.10 the up-call simply increments ul_view , restoring the equality $ul_view = cur_view^r$, and then calls **ApplyRemoval@X**, thus preserving the inductive hypothesis. If $X = G$ then **ApplyRemoval^r@G** increments ul_view (restoring the equality with cur_view^r) and then invokes either **ApplyRemoval@G** or **ApplyRemoval@D**, according as $ul_view > \text{MAGIC_VIEW}$ or not. By Definition 24 $ul_view > \text{MAGIC_VIEW}$ exactly when the constellation is post-critical, which must be the case here when $X = G$, therefore **ApplyRemoval@G** is invoked and **ReplicatedData** is modified the same way in both histories.
4. The notification is for the critical view and $X \neq G$. In this case process X in H invokes the up-call **ApplyJoin@X** and in H^r the up-call **ApplyRemoval^r@X**. By 5.4.10 the **ApplyRemoval^r@X** call increments ul_view , thus restoring the equality $ul_view = cur_view^r$. The up-call proceeds to invoke **ApplyJoin@X**. This modifies **ReplicatedData** the same way as in H .
5. The notification is for the critical view and $X = G$. In this case process G in H invokes the up-call **ApplyJoin@G** and then launches the **APP** thread with **pid** = G . In H^r it invokes **ApplyRemoval@G**. By 5.4.10 the **ApplyRemoval^r@G** call increments ul_view , thus restoring the equality $ul_view = cur_view^r$. The up-call proceeds to invoke **ApplyJoin@G** with **pid** = G exactly as G did in H . This preserves the equality of **ReplicatedData**.

In H^r the **APP** thread is not launched. But since G is an original process in H^r the thread was already launched, with the same parameter, at the beginning of time when the **protStart** procedure was invoked. Since both invocations are asynchronous we may assume that by sheer luck the early invocation of the thread in H^r is delayed so much that the thread starts execution at the exact same point in time in both histories.

- The vector time is reset. This means that the previous vector time is replaced with a vector of zeroes, one per process in **MSet**. This is easily seen to preserve the inductive hypothesis.

At this point we have to take stock of the case where the view installation was the critical one. This is the boundary between the Interim period, where the Interim Hypotheses are in force, and the Convergent period. Following the steps we took so far demonstrates that all the

differences that existed in the state of the process in the two histories have now dissolved. The $\pm G$ difference in **MSet** has been bridged. As a result **vt** converged as well. **PendViewQueue** shed the one record that was different and became equal. **ReceiveSet** and **FwdQueue[]** have gone from being equivalent to being equal, as we have seen.

Since we managed to install the views we know that the self flush height is high namely

$$\text{flush}[X] = \text{cur_view} + v_gap$$

From Lemma 8(8) it follows that $\text{flush_height} = \text{cur_view} + v_gap$. Since we start out with $v_gap > 0$ (otherwise no views are installed) we know by Lemma 8(9) that **WaitSet** must be empty at this point in both histories and therefore equal as well. The only remaining possible difference is in **ContactSet**, in the case $X = G$ only. But this difference cannot exist here because Lemma 8(5) implies that if $Y \in \text{LiveSet} \setminus \text{ContactSet}$ then $\text{flush}[Y] < \text{cur_view} + v_gap$. This would have prevented the views from being installed. As a result the Convergent Hypothesis holds, and the computations have now converged for X .

- the next step is an invocation of the Scan procedure. By Lemma 26, this step preserves the inductive hypothesis and creates no side effects

The last step involves broadcasting all the messages out of **LaunchQueue**.

At this point $v_gap = 0$ and therefore we cannot be in the interim period. Therefore we only need to consider the First Pre-Critical and the Convergent Hypotheses. Under either hypothesis $\text{LaunchQueue}^r = \text{LaunchQueue}$ and by Lemma 25 each call to **protBroadcast** preserves the hypothesis and generates equivalent side effects in both histories. From Lemma 24 it follows that the target set of each side effect is equal to **ContactSet**^r in H^r . \square

Lemma 30. *Suppose that G , $-G$ and D are in the middle of the execution of a pre-critical constellation in H^r . Suppose that the Second Pre-Critical Hypothesis holds. Also suppose that $ul_view = \text{cur_view}^r$ in all three processes. Then executing the **TryToInstall** procedure in all three processes preserves the inductive hypothesis and produces no side effects in either G or $-G$.*

Proof. We follow the execution step by step:

The procedure starts with a loop that goes over the live processes, looking for an impediment to installation of the next view in the form of $\text{flush}[]$ values that are too low. In the situation under consideration we have the same values of cur_view , v_gap , **LiveSet** and $\text{flush}[]$. Therefore all three processes reach the same decision on installation of the pending views. Since **PendViewQueue** is also identical among the processes, the same views get installed.

The next part of the procedure is a loop that installs all the pending views. It goes through the following steps:

- The obsolete messages are removed from **ReceiveSet** and **FwdQueue[]**. Because both sets are identical in all three processes, the same messages are discarded in all three processes and the hypothesis is preserved.
- The next two steps increment cur_view and decrement v_gap . This obviously preserves the inductive hypothesis. It also results in $ul_view = \text{cur_view}^r - 1$.

- The next step pops the head of `PendViewQueue`. This trivially preserves the inductive hypothesis, as well as yielding an identical value for *notification*.
- The next step updates `MSet` according to the type of notification, and applies the change to the replicated application data. Since we are dealing with a pre-critical constellation, the view change must be a removal of a process, and by Definition 24 $cur_view \leq MAGIC_VIEW$. Process D executes `ApplyRemovalr@D` while $\pm G$ execute `ApplyRemovalr@G`. By 5.4.10, this results in all cases in incrementing ul_view , which restores the equality $ul_view = cur_view^r$, and in invoking `ApplyRemoval@D`, which modifies `ReplicatedData` in the same way in all three processes.
- The vector time is reset. This means that the previous vector time is replaced with a vector of zeroes, one per process in `MSet`. This is easily seen to preserve the inductive hypothesis.
- the next step is an invocation of the Scan procedure. By Lemma 26, this step preserves the inductive hypothesis and creates no side effects.

The last step involves broadcasting all the messages out of `LaunchQueue`. By the Second Pre-Critical Hypothesis `LaunchQueue` is empty in $\pm G$, so this step generates no side effects there, while generating a single original message broadcast per message in D . As far as the inductive hypothesis is concerned, for every message in `LaunchQueue` that is broadcast by D the value of $mpkt_out.b$ is incremented and a copy of the stamped message is attached to `BcastWaitSet` in D together with an instability vector. In addition `LaunchQueue` is emptied out in D . These state changes do not violate the Second Pre-Critical Hypothesis, which only requires that `BcastWaitSet` and `LaunchQueue` be empty and $mpkt_out.b$ be zero in $\pm G$. \square

6.6 Proof of the History Equivalence Theorem

We start the proof at the beginning of time. At time zero, each member process is initialized by the `protStart` procedure (see 3.2.5). Direct inspection shows that the state differences between H processes and H^r processes conform to the pre-critical inductive hypotheses. Notice also that after `protStart` is executed in H^r

$$ul_view = cur_view^r = 0$$

It is easy to check that the equality between ul_view and cur_view^r continues to hold as long as H^r executes `CBCAST`. This is because the identity is passed from parent to child, and the only place where either value is changed is in the `TryToInstall` procedure, where these two values are incremented in tandem.

Suppose that the inductive hypothesis holds at constellation label $L = [\ell_t.s.\hat{0}|\hat{0}]$. Both H and H^r have a set of transactions or sub-transactions which share the constellation label L . But in H these transactions are generated by the execution of `CBCAST` procedures as a reaction to triggers, whereas in H^r the transactions are manufactured artificially. We have to show three things. First we must show that if H^r executes the `CBCAST` protocol it will produce transactions that are identical to its observed ones. Then we have to show that if H^r executes the `CBCAST` protocol, then once all the (sub-)transactions in the constellation conclude the inductive hypothesis continues to hold. As a final step we have to show that the triggers of the next constellation in H^r arise naturally from the past behavior of H^r . This guarantees that H^r continues to look like an execution of `CBCAST` and

APP all the way up to the next constellation, and that the inductive hypothesis continues to hold until that point.

We divide our analysis at each stage into cases, depending on the type of constellation, and the type of side effects (See 4.2.1 - 4.2.5) we observe in H within the constellation. In each case we try to deal with all of the inductive hypotheses together. In some cases only some of the inductive hypotheses are relevant. For example, join notifications do not occur pre-critically, by definition.

We also make a repeated use of arguments that show that certain variables that start out equivalent at the start of a constellation end up remaining equivalent at the end. In the Convergent period these variables are required to be identical and not just equivalent, so proving that they remain equivalent is not quite enough. But as long as the procedural parameters that are used by CBCAST to execute each transaction are identical in both histories, the state will remain identical at the end rather than just equivalent. We will point out the few cases where the procedural parameters are not equal, and otherwise gloss over this issue.

6.6.1 Notification constellations

- **Type of Constellation:** A non-critical GMS removal notification of a process R .

H Transactions: One execution of the `protRemove` procedure at each surviving process.

Observed behavior in H^r : Post-critically, the transactions occurring in H^r are equivalent to the H transactions occurring at the same processes. Pre-critically, equivalent transactions occur at the non- G processes, and in addition there are two transactions at $\pm G$. These transactions have the same triggers as the other transactions in the constellation, and their side effects are related to the original transaction at D according to the following cases (see Lemma 12):

- If the original D transaction in H queues a sequence of message multicasts then the $\pm G$ transactions in H^r queue the equivalent message multicasts.
- If the original D transaction in H has no side effects then the $\pm G$ transactions in H^r have no side effects.
- If the original D transaction in H queues a ghost multicast (with or without a succeeding flush multicast) then the $\pm G$ transactions in H^r queue a ghost multicast followed by a flush multicast, of the same height as the original ghost multicast.

Execution of CBCAST in H^r : We follow the execution step by step, dealing with all the inductive hypotheses at once. The first step increments v_gap in all the executions. By the inductive hypothesis v_gap is identical at all compared processes. After incrementation, v_gap remains identical since all the involved processes execute the procedure. Therefore the first step preserves the inductive hypothesis. This step produces no side effects.

The next step adds a record to `PendViewQueue`. In the pre-critical and convergent periods `PendViewQueuer` = `PendViewQueue` and the added record is identical as well, so the First Pre-Critical Hypothesis and the Convergent Hypothesis are preserved. In the interim period `PendViewQueuer` is not identical to `PendViewQueue` but the difference is preserved after the addition of the new record, so the Interim Hypotheses are preserved

as well. Finally, pre-critically PendViewQueue^r is identical at D , G and $-G$ so the Second Pre-Critical Hypothesis is also preserved.

The next two steps remove R from LiveSet and ContactSet . Pre-critically both LiveSet^r and ContactSet^r differ from LiveSet and ContactSet by the addition of $\pm G$, and both sets are identical at D , G and $-G$ in H^r . Since $R \notin \{D, G, -G\}$ in the pre-critical case, the First and Second Pre-Critical Hypotheses are preserved. Post-critically $\text{LiveSet}^r = \text{LiveSet}$ and this property is preserved. Away from G the same is true for ContactSet^r . Therefore the Interim non- G and Convergent Hypotheses are preserved. At G , $\text{ContactSet}^r = \text{LiveSet}$, so the Interim G Hypothesis is preserved as well.

The next step is a loop that stabilizes messages in WaitSet with respect to R . Post-Critically $\text{WaitSet}^r \cong \text{WaitSet}$ in all cases. The loop preserves this relationship, which guarantees the preservation of all the post-critical inductive hypotheses.

Pre-critically, BcastWaitSet^r is empty at $\pm G$ while FwdWaitSet^r is identical at D , G and $-G$ with the exception of a slightly different index at each record. The loop preserves these relationships, which guarantees the preservation of the Second Pre-Critical Hypothesis.

Pre-critically at each non- G process WaitSet^r is very close to being equivalent to WaitSet with the only differences being larger instability vectors. We know that in the pre-critical case $R \notin \{D, G, -G\}$ and therefore removing R from the instability sets in WaitSet^r and WaitSet does not disturb their hypothesized relationship. Finally this fact also guarantees that each instability set will empty out in WaitSet if and only if it empties in WaitSet^r so the loop will always make the same decisions on record removal from WaitSet in both histories. Therefore the First Pre-Critical Hypothesis is preserved as well.

The next step discards $\text{mpkt_in}[R]$. This preserves all the hypotheses.

The next step forwards all the messages out of the forwarding queue $\text{FwdQueue}[R]$. In all periods and at all participating processes $\text{FwdQueue}^r[R] \cong \text{FwdQueue}[R]$, while pre-critically $\text{FwdQueue}^r[R]$ is identical at D , G and $-G$. Therefore the loop proceeds over equivalent or identical messages (depending on which hypothesis we are checking), executing the following steps at each message:

- Popping the head of $\text{FwdQueue}[R]$, yielding a message msg - this preserves all the hypotheses. $\text{msg}^r \cong \text{msg}$ at all processes in all periods. Pre-critically at D , G , and $-G$ the value of msg^r is the same.
- Incrementing $\text{mpkt_out}.f$ - this preserves all the hypotheses.
- Creating $\text{index} = \text{mpkt_out}$. This yields $\text{index}^r = \text{index}$ for all processes in all periods. Pre-critically at D , G and $-G$ this yields:

$$\begin{aligned}\text{index}^r.f(\pm G) &= \text{index}^r.f(D) \\ \text{index}^r.b(\pm G) &= 0\end{aligned}$$

- Creating an instability vector $\text{iset} = \text{mpkt_in}[]$. This yields $\text{iset}^r = \text{iset}$ for all processes in all the post-critical periods. Pre-critically it yields an identical value of iset^r at D , G and $-G$. Pre-critically this also yields the following relation at each

non- G process:

$$\begin{aligned} \text{iset}^r[X].f &= \begin{cases} \text{iset}[X].f & X \neq \pm G \\ \text{iset}[D].f & X = \pm G \end{cases} \\ \text{iset}^r[X].b &= \begin{cases} \text{iset}[X].b & X \neq \pm G \\ 0 & X = \pm G \end{cases} \end{aligned}$$

- Queuing a message multicast containing the message msg and targeted at ContactSet . Since $\text{msg}^r \cong \text{msg}$ this side effect matches the observed side effect in H^r . Since the inductive hypothesis holds at this point, Lemma 24 guarantees that the target set matches the observed target set.
- Adding a record $\langle \text{msg}, \text{index}, \text{iset} \rangle$ to FwdWaitSet . Post critically $\text{FwdWaitSet}^r = \text{FwdWaitSet}$ at every process and the new record is equivalent as well, which preserves the relevant hypothesis.

Pre-critically at D , G and $-G$ in H^r , FwdWaitSet^r is almost identical, except for a zeroed out $.b$ coordinate at each index at $\pm G$. The new record $\langle \text{msg}^r, \text{index}^r, \text{iset}^r \rangle$ exhibits exactly this behavior at the three processes so the Second Pre-Critical Hypothesis is preserved.

Pre-critically at any non- G process WaitSet^r is nearly equivalent to WaitSet , with the addition of $\pm G$ coordinates to iset^r wherever a D coordinate exists in iset . The value of iset^r in the new record in H^r exactly matches the required difference from the value of iset in the new record in H , while $\text{msg}^r \cong \text{msg}$ and $\text{index} = \text{index}$. Therefore the First Pre-Critical Hypothesis is preserved.

In the next few steps we discard $\text{FwdQueue}[R]$, $\text{ghost}[R]$ and $\text{flush}[R]$, actions which preserve all the inductive hypotheses. The one thing to notice is that this follows from the fact that pre-critically $R \notin \{D, G, -G\}$.

In the last step we call CheckFlush which by Corollary 9 preserves the inductive hypotheses. Moreover the same corollary guarantees that the side effects produced by CheckFlush are equal, and therefore equivalent, to the observed side effects. Lemma 24 guarantees that the target set produced by the execution of CheckFlush for each side effect, namely ContactSet^r , is equal to the observed target set.

- **Type of Constellation:** A non-critical GMS join notification of a process J , with parent process K . By definition this must be a post-critical constellation with $J \neq G$.

H Transactions: One execution of the protJoin procedure at each existing process. One execution of the protRun procedure at J . J and K have identical states at the start of the transaction.

Observed behavior in H^r : The transactions at all the processes, including J , are equivalent to the H transactions at the same processes.

Execution of CBCAST in H^r : J and K start with the same state, and the same calls are executed in H^r as in H , at the same processes. Since there is no pre-critical case here,

we only have to verify the preservation of the Interim non- G , Interim G and Convergent Hypotheses. We follow the execution step by step, starting with `protJoin`.

The first few steps increment `v_gap`, add a record to `PendViewQueue`, add J to `LiveSet` and `ContactSet` and create a new and empty entry in `FwdQueue` For the process J . These steps can be easily seen to preserve all three Hypotheses where they apply.

The next step goes over `WaitSet`, and creating an instability coordinate for J whenever such a coordinate exists for the parent process K . Only the forwarding part of the coordinate is copied. The broadcasting part is zeroed. Post-critically, $\text{WaitSet}^r \cong \text{WaitSet}$ so the loop proceeds identically in H and H^r and preserves the applicable inductive hypotheses.

The next three steps create J coordinates in the `ghost[]`, `flush[]` and `mpkt.in[]` vectors. These steps can be easily seen to preserve the applicable hypotheses.

The next two steps create and queue a donation packet to J . Since all the ingredients of the donation vector `donation` are equivalent, the resulting side effect in H^r is equivalent to the corresponding side effect in H , as observed in H^r . The target set of the packet is equal to $\{J\}$ in both histories.

In the last step we call `CheckFlush` which by Corollary 9 preserves the inductive hypotheses. Moreover the same corollary guarantees that the side effects produced by `CheckFlush` are equal, and therefore equivalent, to the observed side effects. Lemma 24 guarantees that the target set produced by the execution of `CheckFlush` for each side effect, namely `ContactSetr`, is equal to the observed target set.

We now look at the execution steps in `protRun`. These steps are taken by J in both H and H^r . Recall from 3.3 that a new process starts out with a state identical to its parent K . Therefore, prior to the execution of `protRun` the states of J in H and H^r conform to the inductive hypothesis for the post-critical state of K , which can vary according as $K = G$ or $K \neq G$. However, if one looks at these two cases in the inductive hypothesis, one sees that they claim the same things, except that in the case $K \neq G$, `ContactSetr = ContactSet` whereas in the case $K = G$, `ContactSetr = LiveSet`. This will make no difference in the end because the `protRun` procedure recomputes `ContactSet` in a way that renders its value identical in both H and H^r as is expected since $J \neq G$.

The first three steps in `protRun` increment `v_gap` and update `PendViewQueue` and `LiveSet`. These steps are easily seen to preserve either the Interim Non- G Hypothesis or the Convergent Hypothesis, as the case may be. The Interim G Hypothesis is mostly preserved except that now `ContactSetr ≠ LiveSet`. This violation is corrected in the next step.

The next step resets `ContactSet` to include the process identifier of J only. This step forces `ContactSet` to be identical in H and H^r . This step preserves the Interim Non- G and the Convergent Hypotheses. If the Interim G Hypothesis applied at the start of the constellation, now the non- G Hypothesis would apply, which is the appropriate hypothesis for J .

All the following steps except the last one rather trivially preserve the appropriate Hypothesis (either the Non- G Hypothesis or the Convergent Hypothesis).

In the last step we call `CheckFlush` which by Corollary 9 preserves the inductive hypotheses. Moreover the same corollary guarantees that the side effects produced by `CheckFlush` are equal, and therefore equivalent, to the observed side effects. Lemma 24 guarantees that the target set produced by the execution of `CheckFlush` for each side effect, namely `ContactSetr`, is equal to the observed target set.

- **Type of Constellation:** The constellation is the critical GMS notification.

***H* Transactions:** One execution of the `protJoin` procedure at each existing process. One execution of the `protRun` procedure at *G*. *G* and *D* have identical states at the start of the transaction.

Observed behavior in *H^r*: The transactions at all the processes, including *G*, are equivalent to their counterparts in *H* with the exceptions that in *H^r* the trigger notification is a removal notification rather than a join notification, and the donation packet queuing event at non-*G* processes in *H* is missing in *H^r*.

Execution of CBCAST in *H^r*: The surviving processes, including *G*, execute the `protRemove` procedure.

Not surprisingly, this case is more subtle than the other cases. First, the processes execute a different call in *H* and *H^r*. Second, the critical notification separates the pre-critical period from the interim period. Therefore, when we argue that the constellation preserves the inductive hypothesis, we are really saying that if the First and Second Pre-Critical Hypotheses hold before the execution of the respective calls, then the Interim *G* and Interim non-*G* Hypotheses hold after the execution.

There is one last complication. Every process that participates in the critical constellation executes exactly one of three procedures. In *H^r*, every process executes the `protRemove` procedure. In *H*, every process except *G* executes the `protJoin` procedure while *G* executes the `protRun` procedure.

The fact that `protRemove` replaces `protJoin` at non-*G* processes explains the observed absence of a queuing of donation packets at those processes in *H^r*.

Each of these calls ends with a call to `CheckFlush`, which is known to preserve all the inductive hypotheses (see Corollary 9). Therefore in the following analysis we are going to ignore the call to `CheckFlush`. We will show that the inductive hypothesis is preserved if each process pauses just before executing this call. Then Corollary 9 together with Lemma 24 will complete the argument.

Due to the fact that different code is executed in *H* and *H^r* we cannot use our usual method of following the parallel execution step by step. Instead we prove this case one state variable at a time. For each variable we show that if the pre-critical hypotheses hold at the start of the execution, then the interim hypotheses (*G* and non-*G*) hold for that variable at the point where `CheckFlush` is invoked.

cur_view

Non-*G* case: Initially *cur_view^r* = *cur_view* according to the First Pre-Critical Hypothesis. Both the `protJoin` and the `protRemove` procedures leave it unchanged, so

it remains equal at the end of the constellation, as required by the Interim non- G Hypothesis.

G case: Initially

$$cur_view^r(G) = cur_view^r(D)$$

according to the Second Pre-Critical Hypothesis. At the same time

$$cur_view^r(D) = cur_view(D)$$

according to the First Pre-Critical Hypothesis, and

$$cur_view(D) = cur_view(G)$$

because G starts life with a state that is identical to the state of its parent (see 3.3). Therefore

$$cur_view^r(G) = cur_view(G)$$

at the start of the constellation. Since the current view is not changed by any of the three procedures, it remains equal at the end of the constellation, as required by the Interim G Hypothesis.

Since this argument will come up repeatedly, we will refer to it as the *sameness argument*.

v_gap

Non- G case: Initially $v_gap^r = v_gap$. Both the `protJoin` and the `protRemove` procedures increment the view gap, so it remains equal as required.

G case: Initially $v_gap^r(G) = v_gap(G)$ by the sameness argument. Both the `protRun` and the `protRemove` procedures increment the view gap, so it remains equal as required.

MSet

Non- G case: Initially $MSet^r = MSet \cup \{\pm G\}$. Neither the `protJoin` procedure nor the `protRemove` procedure change this variable, so the same relation continues to hold at the end of the constellation, as required.

G case: Initially $MSet^r = MSet \cup \{\pm G\}$ as can be easily seen by a slight modification of the sameness argument. Neither the `protRun` procedure nor the `protRemove` procedure change this variable, so the same relation continues to hold at the end of the constellation, as required.

PendViewQueue

Non- G case: Initially $PendViewQueue^r = PendViewQueue$. The `protJoin` procedure appends a $\langle JOIN, G \rangle$ record to `PendViewQueue`, while the `protRemove` procedure adds a $\langle REMOVE, -G \rangle$ record, which is exactly what is required by the Interim non- G Hypothesis.

G case: Initially $PendViewQueue^r = PendViewQueue$ by the sameness argument. The `protRun` procedure appends a $\langle JOIN, G \rangle$ record to `PendViewQueue`, while the `protRemove` procedure adds a $\langle REMOVE, -G \rangle$ record to $PendViewQueue^r$, which is exactly what is required by the Interim G Hypothesis.

LiveSet

Non- G case: Initially $\text{LiveSet}^r = \text{LiveSet} \cup \{\pm G\}$. The protJoin procedure adds G to LiveSet , while the protRemove procedure removes $-G$ from LiveSet^r , so LiveSet^r becomes equal to LiveSet as required by the Interim Non- G Hypothesis.

G case: Initially $\text{LiveSet}^r = \text{LiveSet} \cup \{\pm G\}$ as can be easily seen by a slight modification of the sameness argument. The protRun procedure adds G to LiveSet while the protRemove procedure removes $-G$ from LiveSet^r , LiveSet^r becomes equal to LiveSet as required.

ContactSet

Non- G case: Initially $\text{ContactSet}^r = \text{ContactSet} \cup \{\pm G\}$. The protJoin procedure adds G to ContactSet , while the protRemove procedure removes $-G$ from ContactSet^r , so ContactSet^r becomes equal to ContactSet as required.

G case: It follows from Lemma 8(2) that initially $\text{ContactSet}^r = \text{LiveSet}^r$. The protRemove procedure removes $-G$ from both LiveSet^r and ContactSet^r , keeping them equal.

We already know that at the end of the constellation $\text{LiveSet}^r = \text{LiveSet}$ and therefore we end up with $\text{ContactSet}^r = \text{LiveSet}$, as required by the Interim G Hypothesis.

 $vt[]$

Non- G case: Initially $vt^r[] = vt[] \cup \{[G] = 0, [-G] = 0\}$. Neither the protJoin procedure nor the protRemove procedure make changes to $vt[]$, so the relation remains the same, as required.

G case: Initially $vt^r[] = vt[] \cup \{[G] = 0, [-G] = 0\}$ as can be easily seen by a slight modification of the sameness argument. Neither the protRun procedure nor the protRemove procedure make changes to $vt[]$, so the relation remains the same, as required.

ReplicatedData

Non- G case: Initially $\text{ReplicatedData}^r = \text{ReplicatedData}$. Neither the protJoin procedure nor the protRemove procedure invoke any of the user up-calls (see 2.3.3), so the relation remains the same, as required.

G case: Initially $\text{ReplicatedData}^r = \text{ReplicatedData}$ by the sameness argument. Neither the protRun procedure nor the protRemove procedure invoke any of the user up-calls, so the relation remains the same, as required.

ReceiveSet

Non- G case: Initially $\text{ReceiveSet}^r \cong \text{ReceiveSet}$. Neither the protJoin procedure nor the protRemove procedure make changes to ReceiveSet , so the relation remains the same, as required.

G case: Initially $\text{ReceiveSet}^r \cong \text{ReceiveSet}$ by the sameness argument. Neither the protRun procedure nor the protRemove procedure make changes to ReceiveSet , so the relation remains the same, as required.

FwdQueue[]

Non- G case: Initially $\text{FwdQueue}^r[] \cong \text{FwdQueue}[] \cup \{[G] = \emptyset, [-G] = \emptyset\}$. In H^r the

protRemove procedure forwards all the messages in $\text{FwdQueue}^r[-G]$ and removes it. Since this queue is empty, the overall effect is simply the removal of the empty queue without any side effects. In H the protJoin procedure adds an empty queue for G . Therefore FwdQueue^r becomes equivalent to FwdQueue as required.

G case: Initially $\text{FwdQueue}^r \cong \text{FwdQueue} \cup \{[G] = \emptyset, [-G] = \emptyset\}$ by the sameness argument. In H^r the protRemove removes the empty queue for $-G$ without creating any side effects. In H the protRun procedure adds an empty queue for G . Therefore in this case as well the Interim G Hypothesis holds and no side effects are created.

WaitSet, which includes BcastWaitSet and FwdWaitSet

Non- G case: By the First Pre-Critical Hypothesis, the wait set contains equivalent messages in H^r and H , with a slight difference in their instability sets. Namely that whenever the instability in H has a D coordinate, the instability in H^r has $\pm G$ coordinates in addition. These coordinates have the same f field as the D coordinate, and a zero b field. In H , the protJoin procedure adds the exact same G instability wherever a D instability exists, while in H^r the protRemove procedure removes the $-G$ instability. If a message becomes stable as a result the protRemove procedure removes the message from WaitSet . This however cannot happen because the First Pre-Critical Hypothesis guarantees that the $-G$ instability is accompanied by G and D instabilities. Therefore WaitSet^r becomes equivalent to WaitSet as required by the Interim Non- G Hypothesis.

G case: In this case we have to analyze the two parts of WaitSet separately.

We start with BcastWaitSet . By the Second Pre-Critical Hypothesis BcastWaitSet^r starts out empty. The protRemove procedure does not change this fact, and therefore BcastWaitSet^r ends up empty. On the other hand in H process G executes the protRun procedure which empties BcastWaitSet . So both BcastWaitSet^r and BcastWaitSet end up empty and therefore equivalent, as required by the Interim G Hypothesis.

The case of FwdWaitSet is more complicated. First we verify that FwdWaitSet^r and FwdWaitSet contain equivalent messages. Indeed it follows from the Second Pre-Critical Hypothesis that at the start of the transaction $\text{FwdWaitSet}^r(G)$ and $\text{FwdWaitSet}^r(D)$ contain the same messages. By the First Pre-Critical Hypothesis $\text{FwdWaitSet}^r(D)$ and $\text{FwdWaitSet}(D)$ contain equivalent messages. The protRemove procedure does not remove any messages from $\text{FwdWaitSet}^r(G)$ because any message in $\text{FwdWaitSet}^r(G)$ that has a $-G$ instability also has D and G instabilities. On the other hand $\text{FwdWaitSet}(G)$ is initially equal to $\text{FwdWaitSet}(D)$ and therefore has messages equivalent to the ones in $\text{FwdWaitSet}^r(G)$. The protRun procedure does not remove any messages from $\text{FwdWaitSet}(G)$. Therefore at the end of the constellation $\text{FwdWaitSet}^r(G)$ and $\text{FwdWaitSet}(G)$ contain equivalent messages.

Let msg be any message in $\text{FwdWaitSet}(D)$ at the critical moment. Suppose that its record there is $\langle \text{msg}, \text{index}, \text{iset} \rangle$. By the Self Channel Axiom iset does not contain any D instability and therefore it follows from the First Pre-Critical Hypothesis that the record of msg in $\text{FwdWaitSet}^r(D)$ is equivalent. By the Second Pre-Critical Hypothesis, the record of msg at $\text{FwdWaitSet}^r(G)$ is equivalent

to $\langle \text{msg}, \{f = \text{index}.f, b = 0\}, \text{iset} \rangle$. The `protRemove` procedure does not change this record since it does not contain $-G$ instability and so this remains the record at $\text{FwdWaitSet}^r(G)$ at the end of the constellation. The record of `msg` at $\text{FwdWaitSet}(G)$ starts out equal to the record at $\text{FwdWaitSet}(D)$, and then the `protRun` procedure zeroes out the `index.b` component of the record, leaving it equivalent to the final value of the record at $\text{FwdWaitSet}^r(G)$, as required by the Interim G Hypothesis.

LaunchQueue

non- G case: Initially $\text{LaunchQueue}^r = \text{LaunchQueue}$. Neither the `protJoin` procedure nor the `protRemove` procedure make changes to `LaunchQueue`, so it remains the same, as required.

G case: Initially in H process G inherits its launch queue from D , while in H^r process G has an empty launch queue according to the Second Pre-Critical Hypothesis. The `protRemove` procedure does not change the value of LaunchQueue^r while the `protRun` procedure empties `LaunchQueue`. So LaunchQueue^r becomes identical to `LaunchQueue` as required.

ghost_height and flush_height

Non- G case: Initially $\text{ghost_height}^r = \text{ghost_height}$ and $\text{flush_height}^r = \text{flush_height}$. Neither the `protJoin` nor the `protRemove` procedures make changes to either value (remember that we ignore the `CheckFlush` invocation at the end), so both values remain the same, as required.

G case: Initially G inherits its state from D , so $\text{ghost_height}(G) = \text{ghost_height}(D)$ and $\text{flush_height}(G) = \text{flush_height}(D)$. The Second Pre-Critical Hypothesis implies that $\text{ghost_height}^r(G) = \text{ghost_height}^r(D)$ and $\text{flush_height}^r(G) = \text{ghost_height}^r(D)$. Therefore initially $\text{ghost_height}^r(G) = \text{ghost_height}(G)$. The `protRemove` procedure does not make changes to either variable in H^r while the `protRun` procedure resets the value of flush_height to be equal to ghost_height , therefore at the end of the constellation:

$$\begin{aligned} \text{flush_height}(G) &= \text{ghost_height}(G) = \text{ghost_height}^r(G) = \\ &= \text{ghost_height}^r(D) = \text{flush_height}^r(G) \end{aligned}$$

as required by the Interim G Hypothesis.

mpkt_out

Non- G case: Initially $\text{mpkt_out}^r = \text{mpkt_out}$. The `protJoin` procedure does not change this value in H , while the `protRemove` procedure increments $\text{mpkt_out}^r.f$ for each message in $\text{FwdQueue}[G]$. Since that queue is empty, the value of mpkt_out^r does not change either, and so it remains the same at the end of the constellation, as required by the Interim Non- G Hypothesis.

G case: Initially $\text{mpkt_out}^r.f = \text{mpkt_out}.f$ by the sameness argument and it remains so at the end of the constellation because neither the `protRun` procedure nor the `protRemove` procedure change that value (in the latter case because $\text{FwdQueue}^r[G]$ is empty). Initially $\text{mpkt_out}.b(G) = \text{mpkt_out}.b(D)$ while the Second Pre-Critical Hypothesis implies that initially $\text{mpkt_out}^r.b(G) = 0$. The `protRemove` procedure

does not change that value in H^r while the `protRun` procedure zeroes it in H . Therefore $mpkt_out^r.b(G)$ becomes equal to $mpkt_out.b(G)$.

As a result $mpkt_out^r = mpkt_out$ as required by the Interim G Hypothesis.

mpkt_in[]

Non- G case: We start with the coordinates that exist in H . By the First Pre-Critical Hypothesis, these coordinates have identical values in H and H^r . These coordinates are not touched by either the `protRemove` procedure or the `protJoin` procedure, and so they remain identical at the end of the constellation as required by the Interim non- G Hypothesis.

The $mpkt_in^r[-G]$ coordinate is removed by the `protRemove` procedure, leaving a G coordinate whose value is

$$mpkt_in^r[G] = \{f = mpkt_in^r[D].f; b = 0\}$$

according to the First Pre-Critical Hypothesis. The `protJoin` procedure creates a new G coordinate whose value is

$$mpkt_in[G] = \{f = mpkt_in^r[D].f; b = 0\}$$

Since we have already seen that $mpkt_in^r[D].f = mpkt_in[D].f$, the requirement of the Interim non- G Hypothesis is met.

G case: Initially $mpkt_in^r[] (G) = mpkt_in^r[] (D)$ according to the Second Pre-Critical Hypothesis. The `protRemove` procedure removes the $-G$ coordinate in both at G and D and therefore at the end of the constellation the equality

$$mpkt_in^r[] (G) = mpkt_in^r[] (D)$$

remains valid.

On the other hand the initial relation in H is $mpkt_in[] (G) = mpkt_in[] (D)$ because G is created with the same state as D . The `protJoin` procedure creates a new G coordinate at D with value

$$mpkt_in[G](D) = \{f = mpkt_in[D].f(D); b = 0\}$$

The `protRun` procedure creates a new G coordinate at G with value

$$mpkt_inG = \{f = mpkt_out.f(G); b = 0\}$$

The Self Channel Axiom and Lemma 9 guarantee that at the start of the constellation $mpkt_inD = mpkt_out(D)$. Since G starts out with the same state as D we also have $mpkt_out.f(G) = mpkt_out.f(D)$. Therefore $mpkt_inG = mpkt_in[G](D)$ and the vector as a whole remains identical at G and D .

From the Non- G case we know that $mpkt_in^r[] (D) = mpkt_in[] (D)$ at the end of the constellation. It follows that $mpkt_in^r[] (G) = mpkt_in[] (G)$ at the end of the constellation as required by the Interim G Hypothesis.

ghost and ***flush***

Non- G case: Initially, the First Pre-Critical Hypothesis implies that

$$\begin{aligned} \mathit{ghost}^r[] &= \mathit{ghost}[] \bigcup \{[G] = \mathit{ghost}[D], [-G] = \mathit{ghost}[D]\} \\ \mathit{flush}^r[] &= \mathit{flush}[] \bigcup \{[G] = \mathit{ghost}[D], [-G] = \mathit{ghost}[D]\} \end{aligned}$$

The `protRemove` procedure removes the $-G$ coordinate from both vectors in H^r . The `protJoin` procedure creates a G coordinate in both vectors in H and sets both of them to the value of $\mathit{ghost}[D]$. As a result $\mathit{ghost}^r[]$ and $\mathit{flush}^r[]$ become equal to $\mathit{ghost}[]$ and $\mathit{flush}[]$ as required by the Interim non- G Hypothesis.

G case: Initially $\mathit{ghost}[](G) = \mathit{ghost}[](D)$ and $\mathit{flush}[](G) = \mathit{flush}[](D)$. By the Second Pre-Critical Hypothesis $\mathit{ghost}^r[](G) = \mathit{ghost}^r[](D)$ and $\mathit{flush}^r[](G) = \mathit{flush}^r[](D)$. Using the First Inductive Hypothesis relation between the H and H^r vectors in D we conclude that

$$\begin{aligned} \mathit{ghost}^r[](G) &= \mathit{ghost}[](G) \bigcup \{[G] = \mathit{ghost}[D](G), [-G] = \mathit{ghost}[D](G)\} \\ \mathit{flush}^r[](G) &= \mathit{flush}[](G) \bigcup \{[G] = \mathit{ghost}[D](G), [-G] = \mathit{ghost}[D](G)\} \end{aligned}$$

The `protRemove` procedure removes the $-G$ coordinate from both vectors in H^r . In G , the `protRun` procedure creates a G coordinate in both vectors and sets both of them to $\mathit{ghost_height}(G)$. Since G starts out with the same state as D , we have $\mathit{ghost_height}(G) = \mathit{ghost_height}(D)$. It follows from the Self Channel Axiom and Lemma 10 that at the start of the constellation $\mathit{ghost_height}(D) = \mathit{ghost}D$. Therefore at the end of the constellation

$$\begin{aligned} \mathit{ghost}^rG &= \mathit{ghost}[D](G) = \mathit{flush}^rG \\ \mathit{ghost}G &= \mathit{ghost_height}(G) = \mathit{flush}G \\ \mathit{ghost}[D](G) &= \mathit{ghost}D = \mathit{ghost_height}(D) = \mathit{ghost_height}(G) \end{aligned}$$

and therefore $\mathit{ghost}^r[](G) = \mathit{ghost}[](G)$ and $\mathit{flush}^r[](G) = \mathit{flush}[](G)$ as required by the Interim G Hypothesis.

6.6.2 Message broadcast request constellations

- **Type of Constellation:** A message broadcast request dequeuing event at a process P . We assume (see 2.3.3) that APP at a process P does not issue such a request before its `Main()` function is executed in a separate thread.

H Transactions: An execution of the `protBroadcast` procedure at P .

Observed behavior in H^r : An equivalent transaction at P .

Execution of CBroadcast in H^r : In H^r , process P executes the `protBroadcast` procedure, just as it does in H . In the pre-critical case $P \neq \pm G$. Also notice that $\mathbb{A}^{H^r} = \mathbb{A}^H$ (see 5.4.1) and therefore $\pm G$ do not generate message broadcast requests pre-critically in H^r .

Since the message is not yet stamped we have an actual equality $\text{msg}^r = \text{msg}$ and not just an equivalence.

We start by verifying that in the pre-critical case the Second Pre-Critical Hypothesis is preserved when $P = D$. Processes $\pm G$ do not participate in the constellation in the pre-critical case. It is easy to observe that the `protBroadcast` call does not change any state variables other than `LaunchQueue`, `BcastWaitSet` and `mpkt_out.b`. All of these variables are stipulated by the Second Pre-Critical Hypothesis to be equal to zero at $\pm G$ and so we are done with this case.

To verify the other inductive hypotheses and verify the side effects, we follow the execution step by step.

In the first step, we check if v_gap is zero. If it is not, we append the message to `LaunchQueue`. By the relevant inductive hypothesis (either First Pre-Critical, Interim non- G , Interim G or Convergent Hypothesis) $v_gap^r = v_gap$ and `LaunchQueue` ^{r} = `LaunchQueue`. Since $\text{msg}^r = \text{msg}$ this step is executed identically in H and H^r and preserves the hypotheses.

The next and final step is executed if $v_gap = 0$. Notice that during the interim period $v_gap > 0$ and therefore for this step only the First Pre-Critical and Convergent Hypotheses are relevant. This step contains the following sub-steps:

- The counter `mpkt_out.b` is incremented. $\text{mpkt_out}^r.b = \text{mpkt_out}.b$ and therefore the inductive hypotheses are preserved.
- The next few lines stamp the message. Here the inductive hypotheses requires that the stamping be equivalent, rather than equal, in H and H^r . Since $\text{self}^r = \text{self}$ and $\text{cur_view}^r = \text{cur_view}$ under all the hypotheses these parts of the stamp end up being equal as required.

As for the message vector time, it is computed from the process vector time, and is adjusted at the `self` coordinate. The Convergent Hypothesis is in force when $\text{cur_view} \geq v_{\text{crit}}$ (see 6.4) and it stipulates that all the variables in sight are equal in H and H^r , resulting in $\text{vt}^r[] = \text{vt}[]$. This matches the equivalence requirement (see Definition 22).

The First Pre-Critical Hypothesis is in force when $\text{cur_view} < v_{\text{crit}}$ and it stipulates that

$$\text{vt}^r[] = \text{vt}[] \bigcup \{[G] = 0, [-G] = 0\}$$

The `self` coordinate of vt' is adjusted by $\text{mpkt_out}.b - \text{mpkt_in}[\text{self}].b$. If $P \neq G$ then this adjustment is the same in H and H^r and does not affect the G coordinate. But we already know that in the pre-critical case $P \neq G$, so the hypothesis is preserved in this case.

- In the next step we queue the message multicast to all the members of `ContactSet`. Therefore we have an equivalent side effect to the one in H as observed and Lemma 24 guarantees that the computed target set `ContactSet` ^{r} is equal to the observed target set.

- In the next and last steps, a record for the message is added to **BcastWaitSet**. In the pre-critical case, the message is equivalent and

$$\begin{aligned} \text{index}^r &= \text{mpkt_out}^r = \text{mpkt_out} = \text{index} \\ \text{iset}^r[X].f &= \begin{cases} \text{mpkt_in}^r[X].f = \text{mpkt_in}[X].f = \text{iset}[X].f & \text{if } X \neq \pm G \\ \text{mpkt_in}^r[\pm G].f = \text{mpkt_in}[D].f = \text{iset}[D].f & \text{if } X = \pm G \end{cases} \\ \text{iset}^r[X].b &= \begin{cases} \text{mpkt_in}^r[X].b = \text{mpkt_in}[X].b = \text{iset}[X].b & \text{if } X \neq \pm G \\ \text{mpkt_in}^r[\pm G].b = 0 & \text{if } X = \pm G \end{cases} \end{aligned}$$

as required by the First Pre-Critical Hypothesis.

6.6.3 Message and acknowledgment packet constellations

- **Type of Constellation:** An acknowledgement packet, sent by a process Q in response to an original broadcast of a message m , is received at the originating process P . Note that neither G nor $-G$ broadcast original messages during the pre-critical period or the interim period, therefore we can assume $P \neq \pm G$ in these cases.

H Transactions: A single transaction, with an execution of the **ReceiveAck** procedure at P .

Observed behavior in H^r : Post-critically, or when $Q \neq D$, there is a single transaction at P with an equivalent trigger and equivalent side effects (see Definition 26). Pre-critically when $Q = D$, there are three transactions at P , triggered, in that order, by an acknowledgement packet from $-G$, G and D . The first two have no side effects. The third one has side effects equivalent to the original transaction in H .

Execution of CBCAST in H^r : We divide the analysis into several cases:

- The constellation is post-critical.
- The constellation is pre-critical, with $P, Q \neq D$
- The constellation is pre-critical with $Q \neq D$ and $P = D$
- The constellation is pre-critical, with $Q = D$ and $P \neq D$
- The constellation is pre-critical with $P = Q = D$

In the first three cases, where the constellation is post-critical or $Q \neq D$, the H^r execution is a single invocation of **ReceiveAck** by P . Only in the third of these cases (when $P = D$) is there anything to prove about Second Pre-Critical Hypothesis. We will ignore that part for the moment and return to it later. We will deal with the remaining two cases (where $Q = D$) later as well.

The Convergent case is not entirely trivial here because the trigger is equivalent but perhaps not equal ($\mathbf{p}_{\text{ACK}}\langle m^r \rangle \cong \mathbf{p}_{\text{ACK}}\langle m \rangle$). In fact the trigger is equal but showing that requires the History Equivalence Theorem so we will not rely on this fact and use a weaker argument instead.

To verify the preservation of the First Pre-Critical Hypothesis, Interim non- G Hypothesis, Interim G Hypothesis or Convergent Hypothesis, as the case may be, we go step by step over the procedure.

It starts with a check for the existence of a record for m in **WaitSet**. The record is guaranteed to be there but we have not proved that. However it does follow from the First Pre-Critical and from the two Interim Hypotheses that the record is there in H if and only if it is there in H^r .

The Convergent case is the tricky one here. We know that $m^r \cong m$. If m is pre-critical then it follows from the Convergent Hypothesis that the record is *not* found in either H or H^r . If m is post-critical then $m = m^r$ and the record is found in H if and only if it is found in H^r . Therefore the same decision is made in both histories regardless of which hypothesis is in force, and in the Convergent case if the record is found then the H and H^r triggers are equal.

If the record is not found then the procedure call exits and we are done. With the check successful, the call continues through the following steps:

- The Q entry is removed from the instability set of the message. In the pre-critical case $Q \neq \pm G$, and the Q entry exists in both H and H^r as we have shown. Therefore removing it in both histories preserves the inductive hypothesis.
- If the instability set becomes empty, the record is removed from **WaitSet** and the **CheckFlush** procedure is called. In the post-critical case the instability vector is identical and therefore either empty in both histories or non-empty in both. In the pre-critical case, even though the instability sets are different, they are still empty or non-empty together. Removing the record from **WaitSet** preserves the inductive hypothesis and by Corollary 9, so does invoking **CheckFlush**. Moreover the same corollary guarantees that the side effects produced by **CheckFlush** are equal, and therefore equivalent, to the observed side effects. Lemma 24 guarantees that the target set produced by the execution of **CheckFlush** for each side effect, namely **ContactSet** ^{r} , is equal to the observed target set.

To verify the Second Pre-Critical Hypothesis in the third case, notice that since the message is original, any changes in its record in **WaitSet** only affect the **BcastWaitSet** portion of **WaitSet**. Since the inductive hypothesis only requires that **BcastWaitSet** be empty in $\pm G$, it remains valid until the invocation of **CheckFlush**. In this case we cannot rely on Corollary 9 because $\pm G$ do not participate in the constellation and therefore do not execute **CheckFlush**. Instead we rely on Lemma 8.

If **FwdWaitSet** ^{r} (D) $\neq \emptyset$ when D calls **CheckFlush**, the procedure exists immediately and we are done. If **FwdWaitSet** ^{r} (D) = \emptyset then as we have seen this implies that the set was empty since the beginning of the transaction. This in turn implies that $ghost_height^r(D) = cur_view^r(D) + v_gap^r(D)$. This follows from Lemma 8(9) in the case $v_gap^r(D) > 0$. If $v_gap^r(D) = 0$ then Lemma 8(5) implies that $flush^rD = cur_view^r(D) + v_gap^r(D)$ and the equation follows from Lemma 8(8).

Since $ghost_height^r(D) = cur_view^r(D) + v_gap^r(D)$ the **CheckFlush** procedure does not produce a ghost side effect and as a result does not change the value of $ghost_height^r(D)$.

Therefore the Second Pre-Critical Hypothesis remains valid.

We are left with the two pre-critical cases where $Q = D$.

We start by verifying the preservation of the First Pre-Critical Hypothesis. The execution in H^r consists of three executions of ReceiveAck at P , triggered by the receipt of a packet from $-G$, G and D respectively.

The first two executions remove the $-G$ and G entries from the instability set, but do not empty out the set because it still possesses a D entry. Since CheckFlush is not executed this guarantees that the two executions have no side effects, as expected. However the state at P is now in violation of the First Pre-Critical Hypothesis because m has instability at D without matching instabilities at $\pm G$. The third execution restores the hypothesis by removing D from the instability set. The remaining part of the third execution preserves the inductive hypothesis and produces the expected side effects for the exact same reasons as the previous cases that we already analyzed.

With regard to the preservation of the Second Pre-critical Hypothesis, it is only relevant in the fifth case where $P = D$. Just as in the third case, the changes in WaitSet do not affect the validity of the inductive hypothesis, and it continues to be valid until the moment that D calls CheckFlush. Just as before, Lemma 8 guarantees that this invocation does not produce a ghost broadcast and therefore preserves the Second Pre-Critical Hypothesis.

- **Type of Constellation:** An acknowledgement packet, sent by a process Q in response to the forwarding of a message m , is received at the forwarding process P , in H .

H Transactions: A single transaction, with an execution of the ReceiveAck procedure at P .

Observed behavior in H^r : There are a number of separate cases:

- Post-critically, or when $P, Q \neq D$, there is a single transaction at P with the same trigger and side effects.
- Pre-critically when $P = D$ and $Q \neq D$, there are three transactions at D , G and $-G$ respectively, all triggered by acknowledgement packets from Q for the same forwarded message. The observed side effects at D are equal to the original side effects in H . At $\pm G$ the side effects depend on the D side effects in the following way. If there is a ghost broadcast out of D then there is a same height ghost broadcast followed by a same height flush broadcast out of $\pm G$. If there is no ghost broadcast out of D , then there are no side effects at $\pm G$.
- Pre-critically when $P \neq D$ and $Q = D$, there are three transactions at P , triggered, in that order, by an acknowledgement packet from $-G$, G and D for the same forwarded message. The first two have no side effects. The third one has the same side effect as the original transaction in H .
- Pre-critically when $P = Q = D$, there are nine transactions in total, three each at D , G and $-G$, triggered at each process, in that order, by acknowledgement packets from $-G$, G and D , all for the same forwarded message. The first two transactions in each process have no side effects. The third transaction at D has the side effects as the original transaction at D . The third transaction at each of $\pm G$ has side effects

that depend on the original side effects at D in the same way as in the second case: If there is a ghost broadcast out of D then there is a same height ghost broadcast followed by a same height flush broadcast out of $\pm G$. If there is no ghost broadcast out of D , then there are no side effects for the third transaction at $\pm G$.

Execution of CBCAST in H^r : As before we divide the analysis into the different cases:

- The constellation is post-critical.
- The constellation is pre-critical, with $P, Q \neq D$
- The constellation is pre-critical with $Q \neq D$ and $P = D$
- The constellation is pre-critical, with $Q = D$ and $P \neq D$
- The constellation is pre-critical with $P = Q = D$

The analysis of cases one, two and four is identical to the analysis of similar cases in the case of an acknowledgement for an original message, and we will not repeat it here.

In the third case there are three executions of the ReceiveAck procedure, one execution at each of D , G and $-G$. The analysis of the First Pre-Critical Hypothesis proceeds exactly like the analysis of this case for an original message acknowledgement. As for the Second Pre-Critical Hypothesis, the record for the message in **WaitSet** is in the forwarding part **FwdWaitSet**, which contains the same messages in $\pm G$ as it does in D . Moreover, for each message it contains the same instability set. As a result the execution of the ReceiveAck proceeds in the same way in all three of these processes and the Second Pre-Critical Hypothesis is preserved. By Corollary 9, the invocation of CheckFlush produces side effects that match the observed side effects and Lemma 24 guarantees that these side effects have the observed target set.

In the fifth case there are three executions of the ReceiveAck procedure at each process, triggered by the receipt of an acknowledgement packet from $-G$, G and D , in this order. The analysis of the First Pre-Critical Hypothesis proceeds exactly like the analysis of this case for an original message acknowledgement. As for the Second Pre-Critical Hypothesis, the record for the message m in **WaitSet** is in the forwarding part **FwdWaitSet**, which contains the same messages in $\pm G$ as it does in D . Moreover, for each message it contains the same instability set. As a result, all three executions of the ReceiveAck proceed in the same way in all three of these processes. By Corollary 9, the invocation of CheckFlush preserves the Second Pre-Critical Hypothesis and produces side effects that match the observed side effects and Lemma 24 guarantees that these side effects have the observed target set. Therefore overall the Second Pre-Critical Hypothesis is preserved and the side effects in $\pm G$ match the observed side effects.

- **Type of Constellation:** An original message packet containing message m is received at process P in H .

H Transactions: A single transaction, with an execution of the ReceiveMessage procedure at P .

Observed behavior in H^r : There are two cases

- Post-critically, or when $P \neq D$, there is a single transaction at P with the same trigger and side effect as the original transaction in H .
- Pre-critically, when $P = D$, there are three transactions, one each at D , G and $-G$, with equivalent triggers and side effects as the original transaction in H .

Execution of CBCAST in H^r :

- In the post-critical and $P \neq D$ cases, process P invokes the ReceiveMessage procedure. The execution proceeds through the following steps:

First an acknowledgment is sent back to the sender. This ensures that the side effect in H^r is the expected one.

The next step determines whether the message is original. Since the value of $\text{ORIG}(m)$ is equal in both histories, the determination resolves the same way in both histories, and since the message is original in H it is deemed to be original in H^r as well. As a result $\text{mpkt_in}[\text{ORIG}(m)].b$ is incremented.

In the pre-critical case $\text{ORIG}(m) \neq \pm G$ and therefore incrementing the counter preserves the First Pre-Critical Hypothesis. In the post-critical case, the $\text{mpkt_in}[]$ vector is identical at H and H^r , thus preserving all the relevant hypotheses (Interim G , Interim non- G and Convergent Hypotheses). The Second Pre-Critical Hypothesis is automatically preserved because neither D nor $\pm G$ participate in the constellation in this case.

The next few steps check for duplicates. These checks rely on values, including cur_view , ReceiveSet , the $\text{ORIG}(m)$ coordinate in vt and $\text{VT}(m)$, that are all guaranteed by all the relevant inductive hypotheses to be identical or equivalent at H and H^r . Therefore, the duplicate check yields the same results in both histories.

At this point a comment is due about the Convergent case. The trigger in this case is equivalent ($\mathbf{p}_{\text{MSG}}(m^r) \cong \mathbf{p}_{\text{MSG}}(m)$) but not necessarily equal. Unlike the case of an acknowledgement packet, a non-equal trigger may actually occur in the Convergent period. For that to happen it must be that $\text{VIEW}(m) < v_{\text{crit}}$ (see Definition 22). Since P is convergent, $\text{cur_view} \geq v_{\text{crit}}$. This means that the message is obsolete. Since labeled step 2 of the procedure discards obsolete messages and exits, there is no contamination of the state with non-equal values and the inductive hypothesis holds.

If the message is not a duplicate, the next and last steps are:

- * The message m is added to ReceiveSet . Since both the message m and ReceiveSet are equivalent in both histories, this step preserves the equivalence of ReceiveSet and the inductive hypothesis.
- * The message m is appended to the tail of the sender's forwarding queue. In this case as well $\text{FwdQueue}[\text{sender}]$ is equivalent in H and H^r , and the step preserves the equivalence and the relevant inductive hypothesis.
- * The Scan procedure is invoked. We want to use Lemma 26 to prove that this call preserves all the inductive hypotheses and has no side effects. But the

lemma requires that the ApplyMessage up-call be identical at P in H and H^r . According to 5.4.10 this is the case for $P \neq G$. This covers the pre-critical case since P , as an H process, cannot be equal to G . In the post-critical case the definition of $\text{ApplyMessage}^r@G$ shows that it behaves, post-critically, like $\text{ApplyMessage}@G$, and we are done.

- In the pre-critical case when $P = D$, all three processes D , G and $-G$ invoke the ReceiveMessage procedure. The verification of the First Pre-Critical Hypothesis proceeds in this case exactly as it does in the $P \neq D$ case. As for the Second Pre-Critical Hypothesis, we retrace the execution steps:

First an acknowledgment is sent back to the sender. This ensures that the side effects at G and $-G$ are the expected ones.

The next step determines whether the message is original. Since the value of $\text{orig}(m)$ is equal in all three processes, the determination resolves the same way in all three, and since the message is original at D it is deemed to be original at $\pm G$ as well. As a result $\text{mpkt.in}[\text{orig}(m)].b$ is incremented at all three processes.

Since the $\text{mpkt.in}[]$ vector is identical at all three processes, incrementing the counter at the sender coordinate preserves the inductive hypothesis.

The next few steps check for duplicates. These checks rely on values, including cur_view , ReceiveSet , vt and $\text{vt}(m)$, that are all guaranteed by the Second Pre-Critical Hypothesis to be identical at all three processes. Therefore, the duplicate check yields the same results in all three.

If the message is not a duplicate, the next and last steps are:

- * The message m is added to ReceiveSet . Since both the message m and ReceiveSet are identical in all three processes, this step preserves the Second Pre-Critical Hypothesis.
- * The message m is appended to the tail of the sender's forwarding queue. In this case as well $\text{FwdQueue}[]$ is identical in all three processes and the Hypothesis is preserved.
- * The Scan procedure is invoked. The $\text{ApplyMessage}^r@G$ up-call behaves, pre-critically, like $\text{ApplyMessage}@D$, which in turn behaves like $\text{ApplyMessage}^r@D$ (see 5.4.10). Therefore ApplyMessage behaves the same way in D and in $\pm G$ during the pre-critical period so it follows from Lemma 26 that this call preserves all the inductive hypotheses and has no side effects.

- **Type of Constellation:** A message packet containing a forwarded message m is received at process P from process Q in H .

H Transactions: A single transaction, with an execution of the ReceiveMessage procedure at P .

Observed behavior in H^r : There are several cases:

- Post-critically, or when $P, Q \neq D$, there is a single transaction at P with an equivalent trigger and side effect as the original transaction in H .

- Pre-critically, when $P = D$ and $Q \neq D$, there are three transactions, one each at D , G and $-G$, with triggers and side effects equivalent to the original transaction in H .
- Pre-critically, when $P \neq D$ and $Q = D$, there are three transactions at P , triggered by the receipt of a message packet from D , $-G$, and G , in that order, each with the side effect of an acknowledgement packet being sent to the respective sender. Each transaction is equivalent to the original one in H .
- Pre-critically, when $P = Q = D$, there are nine transactions, three each at D , G and $-G$, triggered by the receipt of a message packet from D , $-G$ and G , in that order, each with the side effect of an acknowledgement packet being sent to the respective sender. Each transaction is equivalent to the original one in H .

Execution of CBCAST in H^r :

- The first two cases proceed just like the equivalent cases of an original message packet, except that the message counter is incremented at the $.f$ coordinate rather than the $.b$ coordinate.
- In the pre-critical case where $P \neq D$ and $Q = D$, the processes D and $\pm G$ do not participate in the constellation and therefore the Second Pre-Critical Hypothesis is automatically preserved. The process P executes the `ReceiveMessage` three times. First it executes the call for the message that it receives from D , and then for the same message being received from $-G$ and G . The first execution proceeds in parallel with the original execution in history H . The analysis of this first execution proceeds in almost the same way as the analysis of the receipt of an original message does. There is an important difference, however. When $mpkt_in[D].f$ is incremented the First Pre-Critical Hypothesis is violated because $mpkt_in[\pm G].f$ is not incremented at the same time. This violation does not, however, affect the rest of the analysis, and remains an isolated violation. This includes the invocation of the `Scan` procedure since Lemma 26 does not require this particular part of the inductive hypothesis.

The subsequent two executions proceed through the following steps:

First, an acknowledgement packet is sent to the respective sender, generating the expected side effect.

The next step increments the forwarded message counter ($mpkt_in[-G].f$ in the second execution and $mpkt_in[G].f$ in the third execution). These steps remove the violation of the First Pre-Critical Hypothesis.

The next steps check for duplicates. These steps cause the call to exit in both executions because the message is obviously a duplicate - it was already placed in `ReceiveSet` and possibly even delivered by the first call.

- In the pre-critical case when $P = Q = D$, each of the processes D , G and $-G$ execute the `ReceiveMessage` three times. First for the message that each receives from D and then for the same message that each receives from $-G$ and then from G . The analysis of the state at D in H and H^r proceeds exactly as in the previous case, where $P \neq D$ and $Q = D$, proving that the First Pre-Critical Hypothesis is preserved in this case. All we have to show is that the three executions at G and

at -G result in the preservation of the Second Pre-Critical Hypothesis. This part of the analysis is just a recap of previous cases. The analysis of the first execution at all three processes (where the message is received from D) proceeds exactly like the same part of the analysis of the case of an original message that is received at D . The next two executions proceed identically at D and at G and -G, resulting in a determination that the message is a duplicate.

6.6.4 Ghost and flush packet constellations

- **Type of Constellation:** A ghost packet of height v is received at a process P from a process Q in H .

H Transactions: A single transaction, with an execution of the ReceiveGhost procedure at process P and no side effects.

Observed behavior in H^r : There are several cases:

- Post-critically, or when $P, Q \neq D$, there is a single transaction at P with the same trigger and no side effects.
- Pre-critically, when $P = D$ and $Q \neq D$, there are three transactions, one each at D , G and -G, each with the same trigger and no side effects.
- Pre-critically, when $P \neq D$ and $Q = D$, there are five transactions at P , triggered by the receipt of identical ghost packets of the same height from -G and then G , followed by flush packets of the same height from -G and then G and finally a ghost packet of the same height from D . This order is induced from the adjustment coordinate of the labels of the triggers. None of these transactions have any side effects.
- Pre-critically, when $P = Q = D$, there are fifteen transactions, five each at D , G and -G, triggered at each process by the receipt of two ghost packets of the same height from -G and then G , followed by flush packets of the same height from -G and then G and finally a ghost packet of the same height from D . None of these transactions have any side effects.

Execution of CBCAST in H^r :

- In the post-critical case P executes the ReceiveGhost procedure which results in raising $ghost[Q]$ to v in both H and H^r . This preserves the relevant one among the Interim non- G , Interim G and Convergent Hypotheses, because all of them stipulate that the $ghost[]$ vector is equal in H and H^r . The procedure does not generate any side effects, which matches the observed lack of side effects in H^r .
- In the pre-critical case where $P, Q \neq D$, process P executes the ReceiveGhost procedure which results in raising $ghost[Q]$ to v in both H and H^r . This preserves the Second Pre-Critical Hypothesis because $P \neq D$ and so no changes occur to the states of either D , G or -G. To see that the First Pre-Critical Inductive Hypothesis is preserved observe that $Q \neq \pm G$ because Q exists in H and $\pm G$ do not, and since $Q \neq D$ by assumption, neither $ghost[\pm G]$ nor $ghost[D]$ is affected. The procedure

does not generate any side effects, which matches the observed lack of side effects in H^r .

- In the pre-critical case when $P = D$ and $Q \neq D$, the three processes D , G and $-G$ each execute the ReceiveGhost procedure, which raises the $ghost[Q]$ level in each process to v . In this case the Second Pre-Critical Hypothesis is preserved because this hypothesis stipulates that $ghost[]$ is equal in all three processes, and they all perform the same change. The First Pre-Critical Hypothesis is preserved for the same reason as in the previous case. The procedure does not generate any side effects, which matches the observed lack of side effects in H^r .
- In the pre-critical case when $P \neq D$ and $Q = D$, the process P executes the ReceiveGhost procedure twice, once for each ghost packet trigger from $-G$ and G , then the ReceiveFlush procedure twice, once for each flush packet trigger from $-G$ and G , and finally the ReceiveGhost procedure once more for the ghost packet trigger from D . Since D and $\pm G$ do not participate in the constellation, the Second Pre-Critical Hypothesis is automatically preserved. At P , the five calls raise the values of $ghost[-G]$, $ghost[G]$, $flush[-G]$, $flush[G]$ and $ghost[D]$ to v . This conforms with the First Pre-Critical Hypothesis. In addition, the calls to ReceiveFlush cause TryToInstall to be invoked.

By the First Pre-Critical Hypothesis the value of v_gap is the same at P in H and H^r at the start of the constellation.

If $v_gap = 0$ at the start of the constellation then by Lemma 8(10) $LaunchQueue = \emptyset$ at P . The first block of TryToInstall does nothing, and the execution of the call skips to the second and last block, where the messages in $LaunchQueue$ are broadcast. Since $LaunchQueue$ is empty the first execution of TryToInstall has no side effects and does not change the process state. The second execution of TryToInstall encounters the same values of v_gap and $LaunchQueue$ as the first execution, and therefore it also does nothing. As a result the whole constellation preserves both pre-inductive hypotheses and produces the observed lack of side effects in H^r .

The case where $v_gap > 0$ is more subtle. First, it follows from the Piggyback Axiom in H that $v \leq cur_view + v_gap$.

By Lemma 10 at the start of the constellation at P in H , $ghost[D] < v$

Furthermore it follows from Lemma 8(5) that at the start of the constellation we have at P , in H :

$$flush[D] \leq ghost[D] < v \leq cur_view + v_gap$$

The First Pre-Critical Hypothesis insures that both inequalities true in H^r as well.

The constellation does not change the value of $flush[D]$, because it does not include the receipt of a flush packet from D . Therefore the test loop at labeled step 1 of TryToInstall fails and the call exits without changing the process state and without any side effects. As a result the same happens in the second call to TryToInstall,

and the constellation as a whole preserves the First Pre-Critical Hypothesis and conforms with the observed behavior of H^r .

- In the pre-critical case when $P = D$ and $Q = D$, the process D executes the ReceiveGhost procedure twice, once for each ghost packet trigger from $-G$ and G , then the ReceiveFlush procedure twice, once for each flush packet trigger from $-G$ and G , and finally the ReceiveGhost procedure once more for the ghost packet trigger from D . The exact same thing happens at G and $-G$, for a grand total of fifteen function calls.

The exact same analysis as in the previous case proves that the First Pre-Critical Hypothesis is preserved with respect to D , and that the resulting side effects in D conform with the observed behavior in H^r (namely, no side effects).

As for the execution at $\pm G$ in H^r , the Second Pre-critical Hypothesis insures that $\pm G$ have the same values of $ghost[]$, $flush[]$ and v_gap as D . Therefore the execution of the constellation in $\pm G$ proceeds in the exact same way as in D , with the same changes to the state and the same lack of side effects.

- **Type of Constellation:** A flush packet of height v is received at a process P from a process Q in H .

H Transactions: A single transaction, with an execution of the ReceiveFlush procedure at process P .

Observed behavior in H^r : There are two cases:

- Post-critically, or when $P \neq D$, there is a single transaction at P with the same trigger and equivalent side effects.
- Pre-critically, when $P = D$, there are three transactions at D , G and $-G$, each triggered by an identical flush packet from Q , and with the side effects at D being equivalent to the ones of the original transaction in H , and with the transactions in G and $-G$ having no side effects.

Execution of CBCAST in H^r :

- Post-critically, or when $P \neq D$, the process P executes the ReceiveFlush procedure in H^r .

The first step in this call increments the value of $flush[Q]$. Post-critically, this step obviously preserves the Interim G , Interim non- G and Convergent Hypotheses, where relevant. Pre-critically, the Second Pre-Critical Hypothesis is preserved because $P \neq D$ and so no changes occur at either D , G or $-G$. To see why the First Pre-Critical Hypothesis is preserved, notice first that in the pre-critical case $Q \neq \pm G$ because Q exists in H and $\pm G$ do not, which is why the $flush[\pm G]$ coordinates are not affected in H^r . Notice also that the value of $ghost[D]$ is not affected in H . Taken together, these two observations explain why the hypothesis is preserved by the first step in this case.

The next and last step the procedure invokes the TryToInstall procedure. By Lemma 29 the TryToInstall call preserves the inductive hypothesis in this case and generates

equivalent side effects. It follows from Lemma 24 that the side effects have the observed target set.

- In the pre-critical case when $P = D$, the analysis of the First Pre-Critical Hypothesis is identical to the analysis of the previous case. For the Second Pre-Critical Hypothesis, there are three transactions in H^r , at D , G and $-G$, each one executing the ReceiveFlush procedure. This call increments the value of $flush[Q]$, which preserves the inductive hypothesis since $flush[]$ is identical in all three processes this case. In the second and last step the invocation of TryToInstall preserves the inductive hypothesis and generates the expected side effects according to Lemma 30.

6.6.5 Donation and co-donation packet constellations

- **Type of Constellation:** A non-critical donation packet is received at a process J from a process P in H .

H Transactions: A single transaction, with an execution of the ReceiveDonation procedure at process J .

Observed behavior in H^r : A single H^r transaction at J that is equivalent to the original H transaction.

Execution of CBICAST in H^r : The process J executes the ReceiveDonation procedure in H^r . Notice that since the donation is not critical then by definition $J \neq G$. Also by definition, donation packets are only sent post-critically. Therefore the only two possible operative inductive hypotheses are the Interim non- G Hypothesis or the Convergent Hypothesis.

The ReceiveDonation call starts with J adding P to $ContactSet$. According to either the Interim Non- G Hypothesis or by the Convergent Hypothesis $ContactSet^r = ContactSet$ and so the first step preserves the inductive hypothesis.

In the next two steps, J sends a co-donation to P in both histories. The contents of $co_donation$ are made up of state variables that are equivalent in H and H^r regardless of whether the Interim non- G or the Convergent Hypothesis is in force. Therefore this side effect is equivalent in both histories.

The next few steps construct the ordered set UNT . It follows from the two Hypotheses and from the fact that $donation^r \cong donation$ that $UNT^r \cong UNT$, since all the ingredients used in the construction of this set are equivalent in both histories. In addition the same hypotheses imply that $mpkt_in^r[] = mpkt_in[]$ and therefore the subsequent loop on the members of UNT invokes the same procedures (either ReceiveMessage or ReceiveAck) in the same order, and with equivalent parameters.

To see that the ReceiveMessage call preserves the inductive hypotheses and generates equivalent side effects, simply follow the reasoning earlier in this proof for the case of a constellation that is triggered by the post-critical receipt of an original message. To see that the ReceiveAck call preserves the inductive hypotheses and generates equivalent side effects, follow the reasoning earlier of this proof for case of a constellation that is

triggered by the post-critical receipt of an acknowledgement in response to an original message.

The last two steps of the call update $ghost[P]$ and $flush[P]$ at J from the donated values of $ghost_height$ and $flush_height$, respectively. The fact that $donation^r \cong donation$ guarantees that the latter two values are the same in H and H^r . As a result $ghost[P]$ and $flush[P]$ remain identical in H and H^r as required by both the Interim non- G and the Convergent Hypotheses.

- **Type of Constellation:** A non-critical co-donation packet is received at a process P from a process J in H .

H Transactions: A single transaction, with an execution of the ReceiveCoDonation procedure at process P .

Observed behavior in H^r : A single H^r transaction at P that is equivalent to the original H transaction.

Execution of CBCAST in H^r : The process P executes the ReceiveCoDonation procedure in H^r . By definition, co-donation packets are only sent post-critically. However it is possible that $P = G$. Therefore the possible operative inductive hypotheses are the Interim non- G Hypothesis, Interim G Hypothesis or the Convergent Hypothesis.

The ReceiveCoDonation call begins with constructing the ordered set UNT . It follows from the three Hypotheses and from the fact that $co_donation^r \cong co_donation$ that $UNT^r \cong UNT$, since all the ingredients used in the construction of this set are equivalent in both histories. In addition the same hypotheses imply that $mpkt_in^r[] = mpkt_in[]$ and therefore the subsequent loop on the members of UNT invokes the same procedures (either ReceiveMessage or ReceiveAck) in the same order, and with equivalent parameters.

To see that the ReceiveMessage call preserves the inductive hypotheses and generates equivalent side effects, simply follow the reasoning earlier in this proof for the case of a constellation that is triggered by the post-critical receipt of an original message. To see that the ReceiveAck call preserves the inductive hypotheses and generates equivalent side effects, follow the reasoning earlier of this proof for case of a constellation that is triggered by the post-critical receipt of an acknowledgement in response to an original message.

The next two steps of the call update $ghost[J]$ and $flush[J]$ at P from the co-donated values of $ghost_height$ and $flush_height$, respectively.

The fact that $co_donation^r \cong co_donation$ guarantees that the latter two values are the same in H and H^r . As a result $ghost[J]$ and $flush[J]$ remain identical in H and H^r as required by the three Hypotheses.

In the last step the TryToInstall procedure is invoked. By Lemma 29, this call preserves the inductive hypotheses and generates equivalent side effects. It follows from Lemma 24 that the side effects have the correct target set, namely $ContactSet^r$.

- **Type of Constellation:** A critical donation packet is received at G from a process P in H .

H Transactions: A single transaction, with an execution of the ReceiveDonation procedure at process G .

Observed behavior in H^r : The original trigger (the receipt of a donation packet) disappears, since we removed that packet in H^r . The queuing of the co-donation packet also disappears, because the co-donation packet is removed as well. What remains are the side effects of all the sub-transactions, and in addition there is a new trigger event for each untimely packet in the \overrightarrow{PD} channel, with the exception of the acknowledgement packets for original messages, since these packets do not get cloned. The new triggers and the existing side effects line up perfectly. By Corollary 6 there is an order preserving one-to-one correspondence between the sub-transactions and the untimely message packets and forwarded acknowledgement packet in the \overrightarrow{PD} channel. We carefully labeled the side effects and the new triggers so that they pair up according to that correspondence to make complete transactions. In addition, there are untimely ghost and flush packets in the \overrightarrow{PD} channel whose clones give rise to additional triggers for which there are no corresponding sub-transactions in H . This is not a problem, these naked triggers simply represent transactions that do not have side effects. Our only burden is to prove that if H^r executes the CBCAST algorithm, then these ghost and flush triggers will indeed produce no side effects and preserve the inductive hypotheses.

Execution of CBCAST in H^r : The critical donation packet is the first post-critical packet from P that G processes in H . We carefully labeled the untimely packets in $\overrightarrow{PG}^{H^r}$ so that they are all processed within the critical donation constellation. Therefore in H^r process G does not process any packets from P in the interval between the critical constellation and the P -donation constellation. As a result at the start of the constellation we have in H^r , according to Lemma 8(7, 8 and 1)

$$\begin{aligned} \text{ghost}[P](G) &= \text{ghost}[P]^{G@l_{v_{\text{crit}}}} \leq \text{ghost_height}_{P@l_{v_{\text{crit}}}} = \\ &= (\text{cur_view} + v_gap)_{P@l_{v_{\text{crit}}}} < v_{\text{crit}} \end{aligned}$$

$$\begin{aligned} \text{flush}[P](G) &= \text{flush}[P]^{G@l_{v_{\text{crit}}}} \leq \text{flush_height}_{P@l_{v_{\text{crit}}}} = \\ &= (\text{cur_view} + v_gap)_{P@l_{v_{\text{crit}}}} < v_{\text{crit}} \end{aligned}$$

Therefore when the P -donation constellation starts, the value of $\text{flush}[P]$ at G is too low for G to have already installed view v_{crit} . Therefore G is still in its interim period (see 6.2), and therefore the operating inductive hypothesis is the Interim G Hypothesis.

All process G does in H^r is process all the clones of untimely packets in \overrightarrow{PD} . In H , process G executes the ReceiveDonation procedure. This means that it adds P to **ContactSet**, then queues a co-donation to P , and only then proceeds to process a similar sequence of packets, with the ghosts and flushes excluded. The addition of P to **ContactSet** does not violate the Interim G Hypothesis. The fact that a co-donation is queued in H but not in H^r is what we expect since we explicitly removed the critical donation packets from H^r .

So before either history processes the first untimely packet the Interim G Hypothesis still holds and the side effects meet our expectations. We have to show that the Donation Sub-Hypothesis also holds at this point. We have already shown that the values of $ghost[P]$ and $flush[P]$ at G are no higher than the critical values of $ghost_height$ and $flush_height$, respectively, at P . This proves that the Donation Sub-Hypothesis holds at that point.

We are going to show, by induction, that up to the i^{th} untimely clone the history H^r looks like an execution of **CBCAST** and the Donation Sub-Hypothesis holds. This means that we assume that in H^r , process G has processed all the clones of untimely packets in channel \overrightarrow{PD} up to, but not including the i^{th} clone. We assume that in H , process G has executed all the sub-transactions that correspond to the same clones, excluding the ghost and flush clones which do not have corresponding sub-transactions. We also assume that at that point the Donation Sub-Hypothesis holds. Let us look now at the i^{th} clone. What does G do with it in H and H^r ? We look at each case separately.

- **The clone is a message packet.** In this case G invokes the `ReceiveMessage` procedure in both histories. The message can be either original or forwarded. We have already analyzed similar cases earlier in the proof (post-critical message receipt) and showed that in both cases the Interim G Hypothesis is preserved. Since the Donation Sub-Hypothesis only differs from the Interim G Hypothesis with respect to variables that are not used by the `ReceiveMessage` procedure, the same analysis holds without change in this case as well.
- **The clone is an acknowledgement packet.** In this case G invokes the `ReceiveAck` procedure in both histories. The packet has to be in response to a forwarded message. We have already analyzed a similar case earlier in the proof (post-critical receipt of a forwarded acknowledgement packet) and showed that the Interim G Hypothesis is preserved. Since the Donation Sub-Hypothesis only differs from the Interim G Hypothesis with respect to variables that are not used by the `ReceiveAck` procedure, the same analysis holds without change in this case as well.
- **The clone is a ghost packet.** In this case G invokes the `ReceiveGhost` procedure in H^r , but there is no corresponding action in H . The procedure produces no side effects, which is what we would expect (since G does not do anything at all in H), but it does increase $ghost[P]$. This deviation is allowed by the Donation Sub-Hypothesis, as long as the value $ghost[P]$ does not exceed the critical value of $ghost_height$ at P . This is guaranteed by Lemma 10, since the packet being processed is untimely and therefore queued by P pre-critically.
- **The clone is a flush packet.** This case is argued similarly to the ghost case as far as state is concerned. However a flush packet may cause side effects. Since G does nothing in H , there must not be any side effects in H^r . For there to be any side effects in H^r the flush value must be high, i.e. it must be equal to $cur_view + v_gap$. But this is impossible since Lemma 10 guarantees that the flush value is at most the critical value of $flush_height$ at P , and Lemma 8 guarantees that the latter value is lower than v_{crit} , while the current value of $cur_view + v_gap$ at G is at least v_{crit} .

We have established that once the process G is done processing all the sub-transactions

(in H) and all the untimely packets (in H^r), the Donation Sub-Hypothesis still holds.

This is the end of the constellation in H^r . What are the values of $ghost[P]$ and $flush[P]$ at G in this history? Since all the untimely packets are now processed, it follows from Lemma 10 that these values are equal to the critical values of $ghost_height$ and $flush_height$, respectively, at P .

The donation transaction is not yet over in H , however. As the last step in the ReceiveDonation procedure, G updates its values for $ghost[P]$ and $flush[P]$ from the donated values of $ghost_height$ and $flush_height$ that it receives as a donation from P . Since the P donation reflects its critical state, this last step restores the Interim G Hypothesis and we are done.

- **Type of Constellation:** A critical co-donation packet is received at a process P from G in H .

H Transactions: A single transaction, with an execution of the ReceiveCoDonation procedure at process P .

Observed behavior in H^r : Like the case of a critical donation, the observed behavior of a critical co-donation constellation in H^r is made up of triggers and sub-transaction side effects that line up perfectly:

- Side effects of the co-donation transaction in H that carry over to H^r .
- Trigger events, made up of the processing events of cloned and zombied packets:
 - * A trigger for each clone and zombie of an untimely packet in the \overrightarrow{DP} channel. Original message packets and flush packets do not get cloned on this channel, so they have no corresponding triggers. Ghost packets get both cloned and zombied, so each of these packets have two corresponding triggers.
 - * A trigger for each clone of a post-critical, pre- P -donation packet in the \overrightarrow{GG} channel. Acknowledgement packets on this channel do not get cloned, so they do not have a corresponding trigger.

According to Lemma 15, if the invocation of the TryToInstall procedure at the end of the ReceiveCoDonation procedure installs the pending views, then UNT is empty and there are no sub-transactions. According to Corollary 7, whatever sub-transactions do exist line up perfectly with the H^r triggers for message and acknowledgement packets. According to Lemma 20 the side effects of TryToInstall line up with a final, post-critical cloned flush packet trigger.

Therefore, if TryToInstall does not install pending views, then all the ghost and flush triggers produce no side effects. If TryToInstall does install the pending views, then there are no triggers other than ghost and flush triggers, and none of them produces any side effects except possibly the last one. Our challenge is to prove that the CBCAST protocol produces exactly those side effects while preserving the inductive hypothesis.

Execution of CBCAST in H^r : When G receives the critical donation packet from P , it adds P to its **ContactSet** and queues the critical co-donation packet to P . Prior to receiving the donation, P is not in G 's **ContactSet** (see Lemma 8(2)) and therefore it does not

queue any packets to P . As a result, the co-donation packet is the first packet that P receives from G in H . Therefore the value of $ghost[G]$ and $flush[G]$ at P remains equal to the original value that P sets when it executes the critical invocation of `protJoin`.

$$ghost[G]_{P@Crit(G \rightarrow P)} = flush[G]_{P@Crit(G \rightarrow P)} = ghost[D]_{P@l_{v_{crit}}} \leq ghost_height_{D@l_{v_{crit}}}$$

Where the last inequality follows from Lemma 8(7). We also know from Lemma 8 that $ghost_height_{D@l_{v_{crit}}} < v_{crit}$ and therefore P does not have a sufficient value of $flush[G]$ at the start of the constellation to allow for the installation of view v_{crit} . As a result the operating inductive hypothesis at the onset of the constellation is the Interim non- G Hypothesis, and from the inequality above it follows that the First Co-Donation Sub-Hypothesis holds as well.

To complete the proof for this constellation, we first use induction to show that the First Co-Donation Sub-Hypothesis holds, and the expected side effects occur, up to the last untimely clone or zombie. Then we use a second induction to deal with the post-critical clones and the `TryToInstall` side effects.

We start by showing that up to the i^{th} untimely clone the history H^r looks like an execution of `CBCAST` and the First Co-Donation Sub-Hypothesis holds. This means that we assume that in H^r , process P has processed all the clones and zombies of untimely packets in channel \overrightarrow{DP} up to, but not including the i^{th} clone or zombie. We assume that in H , process P has executed all the sub-transactions that correspond to the same clones and zombies, excluding the ghost clones and flush zombies which do not have corresponding sub-transactions. Let us look now at the i^{th} clone or zombie. What does P do with it in H and H^r ? We look at each case individually.

- **The clone is a message packet.** In this case P invokes the `ReceiveMessage` procedure in both histories. The packet has to be a forwarded message packet. We have already analyzed similar cases earlier in the proof (post-critical message receipt) and showed that in both cases the Interim Non- G Hypothesis is preserved. Since the First Co-Donation Sub-Hypothesis only differs from the Interim Non- G Hypothesis with respect to variables that are not used by the `ReceiveMessage` procedure, the same analysis holds without change in this case as well.
- **The clone is an acknowledgement packet.** In this case P invokes the `ReceiveAck` procedure in both histories. We have already analyzed a similar case earlier in the proof (post-critical receipt of a forwarded acknowledgement packet) and showed that the Interim Non- G Hypothesis is preserved. Since the First Co-Donation Sub-Hypothesis only differs from the Interim Non- G Hypothesis with respect to variables that are not used by the `ReceiveAck` procedure, the same analysis holds without change in this case as well.
- **The clone is a ghost packet.** In this case P invokes the `ReceiveGhost` procedure in H^r , but there is no parallel action in H . The procedure produces no side effects, which is what we would expect (since P does not do anything at all in H), but it does increase $ghost^r[G]$. This deviation is allowed by the First Co-Donation Sub-Hypothesis, as long as the value $ghost^r[G]$ does not exceed $ghost_height_{D@l_{v_{crit}}}$. This

is guaranteed by Lemmas 10 and 8, since the packet being processed is untimely and therefore queued by D pre-critically.

- **The zombie is a flush packet.** This case is argued similarly to the ghost case as far as state is concerned with the additional fact that

$$\text{flush_height}_{D@l_{\text{crit}}} \leq \text{ghost_height}_{D@l_{\text{crit}}} < v_{\text{crit}}$$

which follows Lemma 8. The inequality implies that in H^r the value of $\text{flush}[G]$ remains low and as a result there are no side effects. This is what we expect since P does nothing in H and therefore does not produce side effects there.

We have established that once the process P is done processing all the pre-critical sub-transactions (in H) and all the untimely packets (in H^r), the First Co-Donation Sub-Hypothesis still holds and all the side effects are as expected. At this point in H^r the process P has already processed the last pre-critical ghost and flush packets from G . Therefore by Lemma 10

$$\begin{aligned} \text{ghost}^r[G] &= \text{ghost_height}^r_{G@l_{\text{crit}}} = \text{ghost_height}^r_{D@l_{\text{crit}}} = \text{ghost_height}_{D@l_{\text{crit}}} \\ \text{flush}^r[G] &= \text{flush_height}^r_{G@l_{\text{crit}}} = \text{ghost_height}^r_{D@l_{\text{crit}}} = \text{ghost_height}_{D@l_{\text{crit}}} \end{aligned}$$

Where the two rightmost equalities in each line follow from the Second Pre-Critical and First Pre-Critical Hypotheses, respectively. We also know that

$$\text{ghost_height}^r_{G@l_{\text{crit}}} \leq \text{ghost_height}^r_{G@\text{Crit}(P \rightarrow G)} = \text{ghost_height}_{G@\text{Crit}(P \rightarrow G)}$$

To see why the left inequality is true, notice that $\text{Crit}(P \rightarrow G) \dot{<} \text{Crit}(G \rightarrow P)$ (this is mediated by the critical co-donation packet) and therefore we can assume by induction that G executes CBCAST up to constellation $\text{Crit}(P \rightarrow G)$. Now the inequality follows from the fact that ghost_height is monotone increasing and $l_{\text{crit}} \dot{<} \text{Crit}(P \rightarrow G)$. The right equality follows from the Interim G Hypothesis.

Taken together these relations prove that the Second Co-Donation Sub-Hypothesis now holds.

We divide the rest of the analysis into two parts. First, assume that the TryToInstall invocation at the end of the ReceiveCoDonation procedure fails to install the pending views.

We will show that up to the i^{th} post-critical clone the history H^r looks like an execution of CBCAST and the Second Co-Donation Sub-Hypothesis holds. This means that we assume that in H^r , process P has processed all the clones of post-critical packets in channel \overrightarrow{GG} up to, but not including the i^{th} clone. We assume that in H , process P has executed all the sub-transactions that correspond to the same clones, excluding the ghost and flush which do not have corresponding sub-transactions. Let us look now at the i^{th} clone. What does P do with it in H and H^r ? We look at each case individually.

- **The clone is a message packet.** In this case P invokes the ReceiveMessage procedure in both histories. We have already analyzed similar cases earlier in the

proof (post-critical message receipt) and showed that in both cases the Interim Non- G Hypothesis is preserved. Since the Second Co-Donation Sub-Hypothesis only differs from the Interim Non- G Hypothesis with respect to variables that are not used by the ReceiveMessage procedure, the same analysis holds without change in this case as well.

- **The clone is an acknowledgement packet.** This case does not occur since post-critical acknowledgement packets do not get cloned.
- **The clone is a ghost packet.** In this case P invokes the ReceiveGhost procedure in H^r , but there is no parallel action in H . The procedure produces no side effects, which is what we would expect (since P does not do anything at all in H), but it does increment $ghost^r[G]$. This deviation is allowed by the Second Co-Donation Sub-Hypothesis, as long as the value $ghost^r[G]$ does not exceed $ghost_height_{G@Crit(P \rightarrow G)}$. This is guaranteed by Lemma 10, since the packet being processed is queued by G before $Crit(P \rightarrow G)$.
- **The clone is a flush packet⁴.** This case is argued similarly to the ghost case as far as state is concerned with the additional fact that

$$flush_height_{G@Crit(P \rightarrow G)} \leq ghost_height_{G@Crit(P \rightarrow G)}$$

which follows from Lemma 8(8). However a flush packet may cause side effects. Since P does nothing in H , there must not be any side effects in H^r .

Suppose that the flush packet does have side effects. For this to happen, the TryToInstall invocation in ReceiveFlush must install the pending views. To do that it is required that for every $Q \in LiveSet^r$ we have $flush^r[Q] = cur_view^r + v_gap^r$. Since the Second Co-Donation Sub-Hypothesis holds, we know that

$$\begin{aligned} LiveSet^r &= LiveSet \\ cur_view^r &= cur_view \\ v_gap^r &= v_gap \\ flush^r[Q] &= flush[Q] \quad \text{for all } Q \in LiveSet \setminus \{G\} \end{aligned}$$

Therefore in H , for any live $Q \neq G$ we have $flush[Q] = cur_view + v_gap$

Since the flush packet had side effects in H^r we know that the packet was $\mathbf{p}_{FLUSH}\langle v \rangle$ with $v = cur_view + v_gap$. Since the original packet was queued by G before it processed the donation from P it follows that

$$v \leq flush_height_{G@Crit(P \rightarrow G)}$$

In H , right before calling TryToInstall in ReceiveCoDonation, process P updates its value of $flush[G]$, setting it to

$$\begin{aligned} flush[G](P) &= co_donation.flush_height = flush_height_{G@Crit(P \rightarrow G)} \geq \\ &\geq v = cur_view + v_gap \end{aligned}$$

⁴Untimely flushes in \overrightarrow{GP} are zombies, but post-critical flushes are clones

It follows that the invocation of TryToInstall in the ReceiveCoDonation procedure does succeed in installing the pending views in H , contrary to our assumptions.

We have established that once the process P is done processing all the post-critical sub-transactions (in H) and all the post-critical packets (in H^r), the Second Co-Donation Sub-Hypothesis still holds and all the side effects are as expected.

At this point in H process P proceeds to update its values of $ghost[G]$ and $flush[G]$:

$$\begin{aligned} ghost[G] &= co_donation.ghost_height = ghost_height_{G@Crit(P \rightarrow G)} \\ flush[G] &= co_donation.flush_height = flush_height_{G@Crit(P \rightarrow G)} \end{aligned}$$

In H^r , P has processed all the post-critical clones, including the last of the post-critical ghost and flush packets from G . It follows from Lemma 10 that at this point

$$\begin{aligned} ghost^r[G] &= ghost_height_{G@Crit(P \rightarrow G)} \\ flush^r[G] &= flush_height_{G@Crit(P \rightarrow G)} \end{aligned}$$

Therefore $ghost^r[G] = ghost[G]$ and $flush^r[G] = flush[G]$ and the Interim Non- G Hypothesis is restored. The last step in H is an invocation of the TryToInstall procedure, that by assumption fails to install the pending views. Therefore it produces no side effects and does not change any state variables.

We are finally left with the case where TryToInstall does succeed in installing the pending views. It follows from Lemma 15 that in this case UNT is empty and no sub-transactions are executed in H . In addition Lemma 20 shows that the last regular packet queued by G prior to processing the donation from P is $k_{last} = \mathbf{p}_{FLUSH}\langle v \rangle$ with $v = cur_view + v_gap$.

It follows that the only post-critical clones in H^r are ghost and flush packets, and their processing, up to k , causes only allowable state changes and generates no side effects (this is shown in exactly the same way as in the previous case, where TryToInstall does not install pending views).

So at last we are at the point where in H process P is about to update its $ghost[G]$ and $flush[G]$ values and invoke TryToInstall, which is going to succeed in installing the pending views. In H^r process P is about to process $k = \mathbf{p}_{FLUSH}\langle v \rangle$, the last cloned packet from G . The Second Co-Donation Sub-Hypothesis still holds.

Since TryToInstall succeeds in installing the pending views, it follows that in H , after P updates $flush[G]$ we have $flush[G] = cur_view + v_gap$. Therefore

$$v = cur_view + v_gap = co_donation.flush_height = flush_height_{G@Crit(P \rightarrow G)}$$

By Lemma 8 we have

$$\begin{aligned} cur_view + v_gap &\geq co_donation.ghost_height = ghost_height_{G@Crit(P \rightarrow G)} \geq \\ &\geq flush_height_{G@Crit(P \rightarrow G)} = cur_view + v_gap \end{aligned}$$

and therefore P also updates $ghost[G] = cur_view + v_gap$.

In H^r the first step in ReceiveFlush sets $flush^r[G] = v = cur_view + v_gap$ and it follows from Lemma 8(5) that at that point we already have

$$ghost^r[G] = v = cur_view + v_gap$$

Therefore, if we hold both H and H^r at the point where both histories are about to invoke the TryToInstall procedure (in the ReceiveCoDonation and the ReceiveFlush procedures, respectively), the Interim Non- G Hypothesis is restored.

It follows from Lemma 29 that the execution of TryToInstall in both histories generates the expected side effects and causes the interim period to end and the Convergent Hypothesis to hold. Lemma 24 ensures that the side effects have the correct target set. This concludes the proof of the theorem and shows that H and H^r perform the same computation.

7 Causality and Progress With The CBCAST Algorithm

7.1 The Goals of the Analysis

To show that the algorithm works as expected, we have to prove two things. One is the preservation of causality, meaning that a message is only delivered at a process after all the messages on which it depends have already been delivered. The second one is the guarantee of progress - that is to say that all messages eventually get delivered. It is well known (see [10]) that progress cannot be guaranteed in the presence of failures, and it follows from [14] that a similar limitation exists in our model of dynamic membership, even without failures. Therefore we can only prove somewhat less than a full guarantee of progress.

7.2 The Causal Order Property and The Progress Property

Definition 27. We say that a message m is **familiar** to a process P if either

- m is delivered at P .
- There is a sequence of processes

$$P_0, P_1, \dots, P_n = P \quad \text{where } n > 0$$

such that

1. For each $i < n$, the process P_i is the parent of process P_{i+1} , i.e. $\exists v_{j(P_{i+1})}(P_i)^{PR} = n_{JOIN}\langle P_{i+1}, P_i \rangle$.
2. n is delivered at P_0 prior to the $v_{j(P_1)}(P_0)^{PR}$ event.

Definition 28. We say that a history H of the CBCAST protocol has the **Causal Order Property** if messages in H are delivered in an order that respects their causal relationships. Technically, this means that if a process P originally broadcasts a message m that is eventually delivered at a process Q then

1. every message that was broadcast by P prior to broadcasting m is already familiar to Q at the delivery of m .
2. every message that is familiar to P at the broadcasting m is already familiar to Q at the delivery of m .

Definition 29. We say that a history H of the **CBCAST** protocol has the **Progress Property** if every message that is originally broadcast by a non-halting process in H eventually becomes familiar to every non-halting process in H .

Theorem 8 (Causal Order Theorem). *The Causal Order Property holds for every conforming history of the **CBCAST** protocol.*

Theorem 9 (Restricted Progress Theorem). *The Progress Property holds for every finite-join conforming history of the **CBCAST** protocol.*

7.3 Causal Order And Progress In Reduced Histories

As a first step in proving the two theorems, we show that if any of the two properties holds in the reduction of a transactional history, then it holds in the original history as well. This allows us to ignore any finite number of process join notifications that may occur in the course of the history. In our proofs we make use of the fact that H and H^r have been labeled using a common label space \mathcal{L} . This allows us to compare the timing of events that occur in the two histories. As usual we denote by G the first joining process in H , with D denoting its parent. We will casually refer to "time" when we technically mean "constellation label". The critical time is the constellation ℓ_{vrit} in the joint label space, when G joins in H .

Definition 30. We say that a message m is **taken up** at process P if the process moves the message m into **ReceiveSet** (this takes place at labeled step 3 of the **ReceiveMessage** procedure). We will see later that the same message can be received by the same process many times, but is only taken up once.

We need the following basic facts:

- By construction H^r and H have the same message broadcast requests ($\mathbb{A}^r = \mathbb{A}$) and those requests are labeled the same way in both histories. This means that the processes in \mathbb{P}^{H^r} originate the same message broadcasts, at the same time, as they do in H .
- We know from the History Equivalence Theorem (Theorem 7) that every process other than G delivers the same messages, at the same time, in both histories; that the same is true for G post-critically; and that pre-critically process G delivers the same messages in H^r that D delivers in H , at the same time. As a result G is familiar, at any post-critical time, with the same messages in both histories. For other processes in H this is true without restriction.
- It follows from the proof of the History Equivalence Theorem that every process other than G takes up the same messages, at the same time, in both histories; that the same is true for G post-critically; and that pre-critically process G takes up the same messages in H^r that D takes up in H , at the same time. See in particular the parts of the proof that relate to receiving message packets, donation packets and co-donation packets. Notice that cloned message packets always result in the message being discarded rather than being taken up.

Theorem 10. *Let H be a transactional history that includes at least one join notification, and let H^r be its reduction. If The Causal Order Property holds in H^r then it holds in H as well.*

Proof. Let P and Q be any processes in H and let m be a message that originates at P and is eventually delivered at Q . Let n be a message is either originated by P prior to originating m or is familiar to P when it originates m .

If P originates n prior to m in H , then P also originates n prior to m in H^r . If P is familiar with n when it originates m in H , it is also familiar with it in H^r when it originates m . Since Q delivers m in H , it also delivers m in H^r . Because the Causal Order Property holds in H^r this implies that Q is familiar with n at the time that it delivers m in H^r . If that time is post-critical, then Q is also familiar with n at the same time in H and we are done. If the time is pre-critical, then $Q \neq G$ (because G does not exist in H pre-critically) and therefore Q is familiar with n at the same time in H regardless. \square

Theorem 11. *Let H be a transactional history that includes at least one join notification, and let H^r be its reduction. If The Progress Property holds in H^r then it holds in H as well.*

Proof. Let P and Q be two processes in H that never halt, and let m be a message that is originated by P . Then P and Q , as processes in H^r are also non-halting, and P originates m in H^r as well. Since the Progress Property holds in H^r , Q eventually becomes familiar with m in H^r . If this happens post-critically or if $Q \neq G$ then Q also becomes familiar with m in H . If it happens pre-critically and $Q = G$ then after the critical time G becomes familiar in H with all the messages that G was familiar with in H^r pre-critically, since it does not halt. Therefore in all cases Q eventually becomes familiar with m in H . \square

Transactional histories are artificial and do not arise naturally the way conforming histories do. In addition, the reduction of a transactional history is rarely transactional. Therefore we need the following lemma.

Lemma 31. *Let H be a conforming history and let $\text{tr}(H)$ be its transactional closure (see the Fault Theorem, Theorem 1). If $\text{tr}(H)$ has the Causal Order Property then so does H . If $\text{tr}(H)$ has the Progress Property then so does H .*

Proof. We start with the Causal Order Property. Suppose that in H a message m originates at process P and is delivered at process Q . Suppose that a message n is either originated by P prior to originating m , or else is familiar to P at the time that it originates m . History $\text{tr}(H)$ contains all the events of H , and by construction an H dequeuing event is processed the same way in $\text{tr}(H)$ as it is in H . Therefore m originates at P and is delivered at Q in $\text{tr}(H)$ and n is originated at P or is familiar to P in $\text{tr}(H)$ prior to the origination of m . Since $\text{tr}(H)$ has the causal order property, n must be delivered at Q prior to the delivery of m , in $\text{tr}(H)$. By construction $\text{tr}(H)$ does not contain any new events that precede existing H events. Therefore n must be delivered at an original H event, which means that n is delivered in H as well. Therefore H has the Causal Order Property.

To prove the Progress Property suppose that in H a message m originates at a non-halting process P and suppose that Q is also a non-halting process, not necessarily distinct from P . Then in $\text{tr}(H)$ both P and Q are non-halting and P originates m . Since $\text{tr}(H)$ has the Progress Property m is delivered at Q in $\text{tr}(H)$, as part of the processing of some event e . If e is an original H event then m is delivered in H and we are done. Otherwise e is a vacuum event that is generated by the vacuum loop. But the vacuum loop is only applied to halting processes, and Q does not halt. This concludes the proof. \square

7.4 Stunting

A *join-free history* is a history H where no processes join. In other words all the processes in \mathbb{P}^H are members of view zero, and all view changes are removals of existing members. The discussion so far implies that if only we could prove that all join-free transactional histories enjoy the Causal Order Property and the Progress Property, then any finite-join conforming history would enjoy these properties as well. We are going to prove exactly that in the next section. As far as the Restricted Progress Theorem is concerned, nothing more is claimed. However, the Causal Order Theorem claims that the Causal Order Property holds for all conforming histories, not only finite-join ones. We plug this gap by introducing the notion of *stunting*.

Definition 31. Let H be a transactional history and let $0 < v < \mathfrak{V}^H$ be any view change in H . Then the **stunting** of H at view v , denoted $H^{<v}$, is the stunted history (see Definition 5) that is obtained from H by taking only that part of H that precedes the view change constellation ℓ_v . We now define the stunting as follows:

$$\begin{aligned}
\mathbb{P}^{H^{<v}} &= \bigcup_{u < v} \mathbb{S}_u^H \\
\mathbb{P}_h^{H^{<v}} &= \mathbb{P}_h^{H^{<v}} \\
\mathfrak{V}^{H^{<v}} &= v \\
\mathbb{S}_i^{H^{<v}} &= \mathbb{S}_i^H \quad \text{for all } i < v \\
\mathbb{K}^{H^{<v}} &= \left\{ k \in \mathbb{K}^H \mid \text{view}(k^{\text{qu}}) < v \right\} \\
\overrightarrow{PQ}^{H^{<v}} &= \overrightarrow{PQ}^H \cap \mathbb{K}^{H^{<v}} \quad \text{for all } P, Q \in \mathbb{P}^{H^{<v}} \\
\mathbb{E}^{H^{<v}} &= \left\{ e \in \mathbb{E}^H \mid \text{view}(e) < v \right\} \\
\prec^{H^{<v}} &= \prec^H \cap \left(\mathbb{E}^{H^{<v}} \times \mathbb{E}^{H^{<v}} \right) \\
\mathbb{F}^{H^{<v}} &= \{ v_i(P) \in \mathbb{F} \mid i < v \} \\
\mathbb{A}^{H^{<v}} &= \left\{ r \in \mathbb{A} \mid r^{\text{PR}} \in \mathbb{E}^{H^{<v}} \right\}
\end{aligned}$$

All the packets and notifications have the same faulting characteristics that they inherit from H . Since H is transactional there are no dropped packets or notifications.

The proof that $H^{<v}$ is a conforming history is tedious but straightforward and we omit it. Moreover, if all the processes in $H^{<v}$ start with the same internal state they had in H then $H^{<v}$ becomes a history of the execution of the CBCAST protocol and at any point in time up to ℓ_v the internal state in the processes in $H^{<v}$ is identical to the internal state of the same processes in H .

Theorem 12. Assume that the Causal Order Property holds for all join-free transactional histories. Then it holds for all conforming histories.

Proof. We have already shown that under the assumption the property holds for all finite-join conforming histories. Let H be an infinite-join conforming history and let P and Q be processes in H . Suppose that P originates a message m and that m is delivered at Q . Assume also that a message n is either originated by P prior to the origination of m or that n is familiar to P at the

time that it originates m . Since H is an infinite-join history, there is a view v for which the view change notification event occurs after Q delivers m .

Look at the v stunting of H . Since $H^{<v}$ has the same state as H up to time ℓ_v and since m is delivered at Q before time ℓ_v , m is also delivered at Q at the same time in $H^{<v}$. The origination of m by P occurs before Q delivers m , so it also occurs before ℓ_v and therefore occurs in $H^{<v}$.

For the same reasons message n is either originated by P prior to m or is familiar to P at the time m is originated, in the stunted history. Since the stunted history is a finite-join conforming history, the Causal Order Property holds there and therefore Q is familiar with n at the time that it delivers m . Therefore the same happens in H and we are done. \square

All we have left to do is prove the Causal Order Theorem (Theorem 8) and the Restricted Progress Theorem (Theorem 9) in the case of join-free transactional histories. For the rest of the paper we will consider only such histories. In particular, we will assume without further comment that every view notification is a removal, and we will not refer to message familiarity (see Definition 27) since it is now synonymous with message delivery. Any definitions (and most importantly, the notion of effective route) will only be assumed to make sense for join-free transactional histories.

In a join-free history of CBCAST the values of LiveSet, ContactSet and MSet have the relations

$$\begin{aligned}\text{LiveSet} &= \text{ContactSet} \\ \text{LiveSet} &\subset \text{MSet}\end{aligned}$$

Therefore we will assume that all live processes are contacted and are members of the current view.

7.5 The Central Lemma

Both theorems rely heavily on the a lemma which we refer to as the Central Lemma, and which we will introduce after a few definitions.

Definition 32. *The **installation gap** of view v at process P is the gap between v and the highest view known to P at the time that v is installed at P . It is the value of v_gap when the view v is installed at process P . More precisely, it is the value of v_gap at labeled step 4 of the TryToInstall procedure when $cur_view = v$. The installation gap of a view v at process P is denoted by $gap_v(P)$.*

Lemma 32. *Let P and Q be processes and suppose that P installs view v when Q is still alive ($Q \in \text{LiveSet}$). Then P must have received a $p_{\text{FLUSH}}(v + gap_v(P))$ from Q prior to installing the view.*

Proof. Looking at the TryToInstall procedure one can verify that $flush[Q] = v + gap_v(P)$ when view v is installed at P . The claim now follows from Lemma 10 and from the monotonicity of $flush[Q]$ (Lemma 8(6)). \square

Definition 33. *Due to forwarding, each message can be received and acknowledged multiple times by a process, but the message is moved into ReceiveSet and FwdQueue[] at most once (See labeled step 1 of ReceiveMessage). Additionally, since a forwarding queue becomes empty after forwarding (see labeled step 6 of the protRemove procedure), a sender will only send a message once to any target. Therefore for any message that is received by a target there is at most one **effective sender***

- namely the sender that managed to get its message packet not just acknowledged but also appended to the forwarding queue of the receiver - and only one **effective packet** that is sent by the effective sender. A message gets forwarded only as a result of a notification of the removal of the effective sender. As a result, for every process that receives a message, there is at most one **effective route** of retransmissions of effective packets leading from the message originator to the process.

Lemma 33. Let n be a message and let

$$\text{ORIG}(n) = R_0 \rightarrow R_1 \rightarrow R_2 \rightarrow R_3 \rightarrow \dots \rightarrow R_k = T \quad \text{where } k \geq 0$$

Be the effective route of n from its originator R_0 to process T . Then for every $0 \leq i \leq k$ the process R_i is a member of view $\text{VIEW}(n)$ and

$$r(R_0) < r(R_1) < \dots < r(R_{k-1})$$

Proof. Messages are only originated or forwarded by members of the message view to members of the message view (see labeled step 2 of the `protBroadcast` procedure and labeled step 4 of the `protRemove` procedure). That proves the first claim. Forwarding is triggered by the removal notification of the effective sender, and view change notifications are only queued to members of that view. That proves the second claim. \square

Lemma 34. When a message is received by a process for the first time it is moved into `ReceiveSet` and `FwdQueue[]`. Therefore for every message n that is received by a process R there is exactly one effective route from the originator of n to R .

Proof. Following the flow of `ReceiveMessage` we see that there are three cases where a message can be discarded without being moved into `ReceiveSet` and `FwdQueue[]`. The second and third cases occur when the message has already been delivered and when the message is still in `ReceiveSet`. Neither of these cases can occur the first time the message is received. The first case occurs when the message is obsolete. To prove the lemma we have to show that in this case as well the message is not received for the first time. Suppose therefore that a process S sent a message packet $\mathbf{p}_{\text{MSG}}\langle n \rangle$ to process R , which then discarded the message as obsolete. Either S is the originator of the message or it forwards n out of its `FwdQueue[]`. Either way there exists in this case an effective route

$$G = G_0 \rightarrow G_1 \rightarrow \dots \rightarrow G_k = S \quad \text{where } k \geq 0 \tag{1}$$

Where $G = \text{ORIG}(n)$ is the originator of n . Notice that $k = 0$ covers the case where $S = G$, i.e. the case where S is the originator of n .

From Lemma 33 we know that $r(G_0) < r(G_1) < \dots < r(G_{k-1})$. In addition $r(G_{k-1}) < r(S)$ because S forwards n (when $k > 0$) as a result of the removal of the effective sender G_{k-1} . We also know that $\text{VIEW}(n) < r(G)$ because G originates n . So for all i , we know that $r(G_i) > \text{VIEW}(n)$ and therefore $G_i \in \mathbb{S}_{\text{VIEW}(n)}$.

By assumption, message n was obsolete when received by R from S . This means that R had already installed view $\text{VIEW}(n) + 1$. It follows from Lemma 32 that process R must have received a $\mathbf{p}_{\text{FLUSH}}\langle > \text{VIEW}(n) \rangle$ packet from every member of $\text{VIEW}(n)$ for which R had not yet received a removal notification. In other words, before R can install $\text{VIEW}(n) + 1$ it must have received either a $\mathbf{p}_{\text{FLUSH}}\langle > \text{VIEW}(n) \rangle$ packet or a $\mathbf{n}_{\text{REM}}\langle P \rangle$ notification from/about every member $P \in \text{VIEW}(n)$.

This is true in particular for the processes in the effective route above, all of whom are members of $\text{VIEW}(n)$.

We know that R does not receive a $\mathbf{n}_{\text{REM}}\langle S \rangle$ notification before installing view $\text{VIEW}(n) + 1$. This is because it receives the message packet $\mathbf{p}_{\text{MSG}}\langle n \rangle$ from S after installing the view, and by the Conforming Packet Axiom this can only happen if R still considers S to be alive. Let i be the index of the first process G_i in the effective route for which R does not receive a $\mathbf{n}_{\text{REM}}\langle G_i \rangle$ notification prior to installing view $\text{VIEW}(n) + 1$.

If $i = 0$, then G_i is the originator of n . Since R does not receive a $\mathbf{n}_{\text{REM}}\langle G_0 \rangle$ notification, it must receive a $\mathbf{p}_{\text{FLUSH}}\langle > \text{VIEW}(n) \rangle$ packet from G_0 . But G_0 only sends such a packet when $\text{cur_view} + \text{v_gap} > \text{VIEW}(n)$. Being the originator of n , process G_0 broadcasts n while $\text{cur_view} = \text{VIEW}(n)$ and $\text{v_gap} = 0$ (messages are not broadcast when $\text{v_gap} > 0$). This means that R receives n from G_0 before receiving the flush packet and therefore before installing view $\text{VIEW}(n) + 1$, so the obsolete $\mathbf{p}_{\text{MSG}}\langle n \rangle$ packet from S is not the first receipt of n .

If $i > 0$, then R receives $\mathbf{n}_{\text{REM}}\langle G_{i-1} \rangle$ prior to installing view $\text{VIEW}(n) + 1$. When that happens, $\text{cur_view} + \text{v_gap} = r(G_{i-1})$ (See labeled step 2 of `protRemove`). It also implies that $r(R) > r(G_{i-1})$. Afterwards, R does not proceed to install $\text{VIEW}(n) + 1$ before receiving $\mathbf{p}_{\text{FLUSH}}\langle \geq r(G_{i-1}) \rangle$ from all the surviving members of $\text{VIEW}(n)$.

Now look at process G_i . it forwards its $\mathbf{p}_{\text{MSG}}\langle n \rangle$ when it receives $\mathbf{n}_{\text{REM}}\langle G_{i-1} \rangle$. It then places n in its `WaitSet`, together with an instability set that includes R (since $r(R) > r(G_{i-1})$). It then waits for `WaitSet` to clear before sending its first $\mathbf{p}_{\text{FLUSH}}\langle \geq r(G_{i-1}) \rangle$. We know that this flush packet is eventually sent by G_i , by definition of i , so n does indeed leave `WaitSet`. We also know that the flush packet is sent while G_i still considers R to be alive, according to the Process Liveness Axiom. Therefore n must leave `WaitSet` as a result of G_i receiving a $\mathbf{p}_{\text{ACK}}\langle n \rangle$ packet from R . This means that R receives n from G_i before installing view $\text{VIEW}(n) + 1$, so the obsolete $\mathbf{p}_{\text{MSG}}\langle n \rangle$ packet from S is not the first receipt of n in this case either. \square

Lemma 35 (Central Lemma). *If process P installs view v_m with $\text{gap}_{v_m}(P) = 0$, then any message n with $\text{view}(n) < v_m$ that is received by P is also received by all other processes that install view v_m .*

We will prove the central lemma a little later on. First, we use it to prove the Causal Order Theorem and the Restricted Progress Theorem.

Proof of the Causal Order Theorem. We start with a note on message origination and broadcasting. The act of originating a message requires two separate steps whenever $\text{v_gap} > 0$. First, placing the message on `LaunchQueue` (see labeled step 1 of the `protBroadcast` procedure), and then later when v_gap becomes zero, queuing packets containing the message (see labeled step 5 of the `TryToInstall` procedure). In the context of this proof, we are talking about the second step whenever we talk about *broadcasting* a message. This definition of the term "broadcast" can create phantom causal relationships between messages. Specifically, if message n is delivered after message m is placed on `LaunchQueue` but before a packet containing m is queued, we consider m to be dependent on n , though they are obviously independent. This does not invalidate our arguments, of course, but it does imply that the algorithm serializes message delivery to a greater degree than seems necessary. This suboptimal behavior is inherent in the protocol, because every view change is a global synchronization point that generates excess serialization.

Let P and Q be processes, and let m be a message that is broadcast by P and is delivered at Q . We have to demonstrate parts 1 and 2 of Definition 28. Remember that message familiarity is now synonymous with message delivery.

We start by observing that whenever a process P broadcasts a message n , the message is delivered at P itself before it processes any further notifications from **GMS**. This follows from the Self Channel Axiom. This axiom only implies that n is *received* by P prior to processing any further notifications from **GMS**. However, when n is received by P it is immediately deliverable. This is easy to verify when you notice that P stamps n with its own vector time before broadcasting it. Moreover, the `ReceiveMessage()` procedure includes a call to `Scan()`, resulting in an immediate delivery of n . As a result, if P broadcasts n before broadcasting m , and if $\text{VIEW}(n) < \text{VIEW}(m)$, then n is familiar to P when it broadcasts m , and we can lump this case under part 2 of definition 28. This leaves the case $\text{VIEW}(n) = \text{VIEW}(m)$. In this case we have $\text{VT}(n) < \text{VT}(m)$ because n gets stamped by P before m does.

When P broadcasts m , it has view

$$\begin{aligned} \text{cur_view} &= \text{VIEW}(m) \\ \text{v_gap} &= 0 \end{aligned}$$

and therefore $\text{gap}_{\text{VIEW}(m)}(P) = 0$.

Also, since m is delivered at Q we know that Q installs $\text{VIEW}(m)$. Therefore P and Q meet the requirements of the Central Lemma. We divide the messages that are broadcast by P or delivered at P prior to the broadcast of m into the following subsets:

1. For each $0 < k < \text{VIEW}(m)$, the set

$$\mathfrak{M}_k = \{n \mid \text{VIEW}(n) = k \text{ and } n \text{ is delivered at } P\}$$

2. All the messages n with $\text{VIEW}(n) = \text{VIEW}(m)$ and $\text{VT}(n) < \text{VT}(m)$

Our previous observation shows that these sets cover all the messages n of both part 1 and part 2 of Definition 28. The messages in the second category must be delivered at Q prior to the delivery of m (see section 5.1 of [6]). For a given $0 < k < \text{VIEW}(m)$, the Central Lemma implies that all the messages in \mathfrak{M}_k are received by Q .

Suppose that there is a message in \mathfrak{M}_k that is not delivered at Q , and let n be such a message with a minimal vector time. Let n' be any message, not necessarily in \mathfrak{M}_k , such that $\text{VIEW}(n') = k$ and $\text{VT}(n') \prec \text{VT}(n)$. Because $n \in \mathfrak{M}_k$, n is delivered at P . Therefore n' must be delivered at P before n is delivered (see section 5.1 of [6]), and therefore $n' \in \mathfrak{M}_k$. By the minimality of n , we know that n' is delivered at Q . Look at the latest of the following points in time:

- The point at which n enters `ReceiveSet(Q)` (at labeled step 3 of `ReceiveMessage`)
- For each n' with $\text{VIEW}(n') = k$ and $\text{VT}(n') \prec \text{VT}(n)$, the point at which n' is delivered at Q (labeled step 1 of `Scan`)
- The point at which Q installs view k (labeled step 4 of `TryToInstall`)

This latest point in time has the following two properties. First, $Scan()$ is called shortly after that point, as can be verified by tracing each of the three code locations. Second, the message n is present in $ReceiveSet(Q)$ at that point. To see that, notice that by definition n must have already entered $ReceiveSet(Q)$ (because of the first point in time), so we only have to show that n has not yet been removed. There are two places where n can be removed. The first is labeled step 1 of $Scan$, which is where n is delivered. Since all the above points in time must occur before n can be delivered, this case can be excluded. The second is labeled step 2 of $TryToInstall$, where n is removed as an obsolete message. But all the points in time that we listed must occur before Q installs a view that is higher than k , so this case can be excluded as well. Therefore the message n is present in $ReceiveSet(Q)$ at the latest point in time. Moreover, n is deliverable at this point because Q has already installed view k and all the messages of lower vector time have been delivered. Therefore the imminent call to $Scan()$ will result in the immediate delivery of n to Q (in the case of labeled step 1 of $Scan$ it may happen even earlier, within the loop). Due to their low view, the messages in \mathfrak{M}_k must be delivered at Q before the installation of $VIEW(m)$ which in turn takes place before the delivery of m . \square

In order to prove the Restricted Progress Theorem, we need the following lemma.

Lemma 36. *Non-halting processes install all the views.*

Proof. The proof depends on the finiteness condition in a fundamental way. Let v_L be the last view. By definition, the non-halting processes are exactly the members of that view. Let P be a non-halting process. At some point in time P dequeues a notification of the last view which by assumption is a notification $\mathbf{n}_{\text{REM}}\langle X \rangle$ of the removal of the last halting process X .

When P processes that notification it forwards all the messages in $\text{FwdQueue}[X]$ in order, and moves them to its WaitSet . Thereafter, as long as P does not install view v_L , P does not add any new messages to WaitSet . This is because messages are added to WaitSet only after broadcasting or forwarding a message. But since $v\text{-gap} > 0$ during the period at hand, P does not broadcast any messages, and since no further view change notifications occur, P does not forward any messages either. All the messages in WaitSet , having been sent by the non-halting process P , are eventually acknowledged by all the non-halting processes.

Since P has already received notice of the removal of all the halting processes, this implies that all the messages in WaitSet eventually stabilize and therefore WaitSet empties. Once that happens, P sends a $\mathbf{p}_{\text{FLUSH}}\langle v_L \rangle$ packet to all the non-halting processes. This means that every non-halting process eventually receives a $\mathbf{p}_{\text{FLUSH}}\langle v_L \rangle$ packet from all the non-halting processes. Once that happens, all the views up to and including v_L are installed immediately. \square

Proof of the Restricted Progress Theorem. Let P and Q be two non-halting processes and suppose P broadcasts a message m . A join-free history can only have a finite number of view changes so we have to show that m is eventually delivered at Q .

Let v_L be the last view. By Lemma 36, both P and Q install v_L , and by necessity they do it with $\text{gap}_{v_L}(P) = \text{gap}_{v_L}(Q) = 0$. By the Central Lemma this implies that all messages that were received by P prior to view v_L are also received by Q and vice versa. More generally, all the non-halting processes receive the same messages prior to installing view v_L .

To see that the same happens with messages that are received after installing view v_L , observe that since view v_L is comprised exclusively of non-halting processes, all messages sent in view v_L are broadcast by non-halting processes and therefore must be received by all members of v_L . Therefore, all the non-halting processes receive the same messages.

We will show now that non-halting processes also deliver the same messages. Let n be a minimal message is delivered at a non-halting process P but not delivered at a non-halting process Q . We know by the discussion above that n is received by Q , and therefore by Lemma 34 n enters Q 's **ReceiveSet**. We know by the Causal Order Theorem all the messages that message n depends on are delivered at P and by minimality are also delivered at Q . Why would n not be delivered? Looking at the Scan and TryToInstall procedures, we see that n enters **ReceiveSet** when $\text{cur_view} \leq \text{VIEW}(n)$, and n is not delivered as long as Q does not install $\text{VIEW}(n)$. In addition n is removed from **ReceiveSet** without being delivered if it has not been delivered by the time Q installs $\text{VIEW}(n) + 1$. From lemma 36 we know that Q installs view $\text{VIEW}(n)$. Look at the latest of the following three points in time:

1. Q has received the effective packet $\mathbf{p}_{\text{MSG}}(n)$, and has just placed n in **ReceiveSet**. (Occurs at the conclusion of labeled step 3 of **ReceiveMessage**.)
2. Q has just delivered the last message that n depends on. (Occurs at the conclusion of labeled step 1 of **Scan**.)
3. Q has just installed $\text{VIEW}(n)$. (Occurs at labeled step 4 of **TryToInstall**.)

When that latest moment occurs, two conditions are met. First, Q has $\text{cur_view} = \text{VIEW}(n)$, because at least the third moment occurs when $\text{cur_view} = \text{VIEW}(n)$, and none of the other two moments can happen after a higher view is installed. Second, the message n is in **ReceiveSet** because it enters the set at the first moment and neither way out of the set (delivery of n or installation of a higher view) is available until after all three moments in time have occurred. As a result, the message n becomes deliverable at that moment. Looking at the relevant code locations, one can see that the **Scan** procedure is either invoked or (in the second case) continues to be executed, resulting in the delivery of n .

We are almost done. We know that all the non-halting processes deliver the same messages, but we have to show that they deliver *all* the messages that originate from non-halting processes. To do that, we will show that every process delivers all the messages that it originates itself. Since all non-halting processes deliver the same messages, this implies the desired result.

Suppose that the non-halting process P originates a message n . Since P is non-halting it receives n and by Lemma 34 it must be placed in P 's **ReceiveSet**. As we already observed in the proof of the Causal Order Theorem, when n is placed in **ReceiveSet** of its own originator it is immediately deliverable, because its $\text{VT}(n)$ is derived from P 's vector time in exactly the fashion that makes it deliverable. Since message placement in **ReceiveSet** is followed by a call to **Scan()** (labeled steps 3 and 5 of **ReceiveMessage**), n is delivered at P and we are done. \square

7.6 Proving the Central Lemma

We start with a technical lemma that we will use repeatedly in the proof of the Central Lemma.

Lemma 37. *Let A , B , D and F be processes (not necessarily distinct), and let n be a message, meeting the following criteria:*

1. *Process D sends an effective $\mathbf{p}_{\text{MSG}}\langle n \rangle$ packet to process F as it broadcasts or forwards n .*
2. *Process B is a member of $\text{VIEW}(n)$.*
3. *$r(D) < r(F)$.*
4. *Process A receives a $\mathbf{p}_{\text{FLUSH}}\langle f \rangle$ packet from process F , with $r(D) \leq f < r(B)$*

Then process B receives n before process F queues a $\mathbf{p}_{\text{FLUSH}}\langle f \rangle$ packet to process A and before B dequeues a notification of view $r(F)$.

Proof. Original messages are broadcast to all the members of the message view (see labeled step 2 of the `protBroadcast` procedure. When no processes join, $\text{ContactSet} = \text{LiveSet}$ and since $\mathbf{v_gap} = 0$ here, $\text{MSet} = \text{LiveSet}$), and are forwarded to all the surviving members of the message view (see labeled step 4 of `protRemove`). By condition 2 process B is a member of $\text{VIEW}(n)$ and condition 4 implies that $r(D) < r(B)$, so process D never receives a $\mathbf{n}_{\text{REM}}\langle B \rangle$ notification. Therefore, when D broadcasts or forwards message n , it queues a $\mathbf{p}_{\text{MSG}}\langle n \rangle$ packet to process B . By condition 1 we know that D also queues a $\mathbf{p}_{\text{MSG}}\langle n \rangle$ packet to F , and that this packet is effective. That means that when process F receives the packet, it appends n to its forwarding queue, with D as the sender (see labeled step 4 of `ReceiveMessage`). Condition 3 implies that F receives a $\mathbf{n}_{\text{REM}}\langle D \rangle$ notification. We divide the rest of the proof into two cases. The easy case is when n is still in the forwarding queue when the $\mathbf{n}_{\text{REM}}\langle D \rangle$ notification is dequeued by F . The harder case is when n has already been removed from the forwarding queue.

If n is still in the forwarding queue, then F forwards n to all the surviving members of view $\text{VIEW}(n)$, which include B because $r(D) < r(B)$. It then moves n to its wait set, together with an instability set that includes B (see labeled step 6 of `protRemove`). The moment that F dequeues the removal notification of D is also the first point in time at which $\text{cur_view} + \mathbf{v_gap} \geq r(D)$ in F (see labeled step 1 of `protRemove`). Therefore, the flush packet that F sends to A according to condition 4 is queued later (see labeled step 2 of `CheckFlush`). Since flush packets are only queued when WaitSet is empty (see condition in ambient block of same location), the flush packet must be queued after message n has left the wait set, i.e. message n must have stabilized in the meantime. But process B was initially in the instability set of message n . Message n can stabilize with respect to process B either by receipt of a removal notification for process B , or by receipt of an acknowledgment of message n by B . In the latter case B must send the acknowledgment to F before becoming aware of view $r(F)$ and we are done. The former case is impossible because condition 4 implies that the flush packet was sent before process F dequeued a notification of the removal of process B .

If n is not in the forwarding queue when a $\mathbf{n}_{\text{REM}}\langle D \rangle$ notification is received by F , it means that it has already been removed. The only possible cause of removal is the installation, by F , of view $\text{VIEW}(n) + 1$ (see labeled step 3 of the `TryToInstall` procedure). Let

$$e = \text{VIEW}(n) + 1 + \mathbf{gap}_{\text{VIEW}(n)+1}(F)$$

F installs view $\text{VIEW}(n) + 1$ only after receiving $\mathbf{p}_{\text{FLUSH}}\langle \geq e \rangle$ from all the surviving members of view $\text{VIEW}(n)$ (see lemma 32). In the case at hand process D is one of the surviving members, because by our assumption F receives the $\mathbf{n}_{\text{REM}}\langle D \rangle$ notification after installing the new view. This also

implies that $e < r(D)$ and therefore, by condition 4, $e < f$, and therefore F sends the $\mathbf{p}_{\text{FLUSH}}\langle f \rangle$ packet to A after installing view $\text{VIEW}(n) + 1$.

If we are lucky, then D sends the effective $\mathbf{p}_{\text{MSG}}\langle n \rangle$ packet before sending $\mathbf{p}_{\text{FLUSH}}\langle \geq e \rangle$ to F . In that case n enters the wait set of D with process B in its instability set, and it must exit the wait set before the flush is sent. Since $r(D) < r(B)$, this can only happen if B receives and acknowledges n . Process B acknowledges the receipt of n before D sends a $\mathbf{p}_{\text{FLUSH}}\langle \geq e \rangle$ packet to F , which is received by F before it installs view $\text{VIEW}(n) + 1$, which in turn happens before F sends a $\mathbf{p}_{\text{FLUSH}}\langle f \rangle$ packet to A . Process B acknowledges the receipt of n before becoming aware of $r(D)$ and $r(D) < r(F)$, so the acknowledgment is sent before B is aware of the death of F and so we are done in this case.

If we are not lucky, then D sends a $\mathbf{p}_{\text{FLUSH}}\langle \geq e \rangle$ to F and only later sends an effective packet $\mathbf{p}_{\text{MSG}}\langle n \rangle$ to F . Let

$$D_0 \rightarrow D_1 \rightarrow \dots \rightarrow D_k = D \rightarrow F \quad \text{where } k \geq 0$$

Be the effective route of the message n from its originator D_0 through D to F . Let i be the smallest integer such that D_i sends a $\mathbf{p}_{\text{FLUSH}}\langle \geq e \rangle$ packet to F before sending an effective $\mathbf{p}_{\text{MSG}}\langle n \rangle$ packet along the effective route. Then $i > 0$ because the originator D_0 broadcasts n while $\mathbf{v_gap} = 0$, so it could not have sent a $\mathbf{p}_{\text{FLUSH}}\langle \geq e \rangle$ beforehand, since $e > \text{VIEW}(n)$. Process D_i forwards n along the effective route after receiving a $\mathbf{n}_{\text{REM}}\langle D_{i-1} \rangle$ notification, which arrives, by the definition of i , after D_i already sent a $\mathbf{p}_{\text{FLUSH}}\langle \geq e \rangle$ packet to F . This implies that $r(D_{i-1}) > e$, which in turn implies (by lemma 32) that F does not install view $\text{VIEW}(n) + 1$ before receiving a $\mathbf{p}_{\text{FLUSH}}\langle \geq e \rangle$ from D_{i-1} . By definition, D_{i-1} must send the effective $\mathbf{p}_{\text{MSG}}\langle n \rangle$ along the effective route before sending the $\mathbf{p}_{\text{FLUSH}}\langle \geq e \rangle$ packet. In addition, by lemma 33 $r(D_{i-1}) < r(D) < r(B)$ so D_{i-1} sends message n to process B and when it moves n to its wait set it includes process B in the instability set of n . As a result, D_{i-1} cannot send a $\mathbf{p}_{\text{FLUSH}}\langle \geq e \rangle$ packet to F before n stabilizes with respect to process B , and since $r(D_{i-1}) < r(B)$, this stabilization can only happen as a result of a receipt of a $\mathbf{p}_{\text{ACK}}\langle n \rangle$ packet from B . So in this case as well B receives and acknowledges message n , and it must happen before F installs view $\text{VIEW}(n) + 1$, which in turn happens before F sends a $\mathbf{p}_{\text{FLUSH}}\langle f \rangle$ packet to A . In addition B must send the acknowledgment of receipt of n before becoming aware of the death of D_{i-1} and since $r(D_{i-1}) < r(D) < r(F)$ this means that B sends the acknowledgment before becoming aware of view $r(F)$. So we are done in this case as well. \square

Proof of the Central Lemma. Assume that there is a message n with $\text{VIEW}(n) < v_m$ that is received by process P , and let Q be some process that installed view v_m . Denote $v_n = \text{VIEW}(n)$. Let $S = \text{ORIG}(n)$, the process that originally broadcast n . Then there is an effective route (see Definition 33 and Lemma 34) of $\mathbf{p}_{\text{MSG}}\langle n \rangle$ packets:

$$S \rightarrow R_1 \rightarrow R_2 \rightarrow R_3 \rightarrow \dots \rightarrow R_k \rightarrow P \quad \text{where } k \geq 0 \quad (2)$$

To streamline our arguments, we will occasionally refer to S as R_0 and to P as R_{k+1} . By Lemma 33 all the processes in the effective route must be members of v_n and

$$r(S) < r(R_1) < \dots < r(R_k)$$

By assumption, P installs view (v_m) , and $\mathbf{gap}_{v_m}(P) = 0$. From Lemma 32 we know that in order to install view v_m process P must first receive a $\mathbf{p}_{\text{FLUSH}}\langle v_m + \mathbf{gap}_{v_m}(P) \rangle$ packet from each member of LiveSet . It is easy to check that $\text{LiveSet} = \text{MSet}$ whenever $\mathbf{v_gap} = 0$. Therefore P must receive

a $\mathbf{p}_{\text{FLUSH}}\langle v_m \rangle$ packet from every member of $\text{view}(v_m)$. We will use that fact several times in the argument below.

$R_{k+1} = P$ is a member of view v_m . Let i be the smallest integer such that R_i is a member of v_m . We will divide the proof to two cases: $i > 0$ and $i = 0$.

We start with the case $i > 0$.

Since $i > 0$ there exists a process R_{i-1} that sends an effective packet $\mathbf{p}_{\text{MSG}}\langle n \rangle$ to R_i , and R_{i-1} is not a member of view v_m . Since both are members of view v_n and R_i is a member of view v_m , $r(R_{i-1})$ is in the membership interval of R_i and so it must receive a $\mathbf{n}_{\text{REM}}\langle R_{i-1} \rangle$ packet from the membership service. We know that $v_n < v_m < r(R_i)$. We look at the following cases:

Case I: $r(Q) \leq r(R_i)$.

We know that Q installs view v_m , and by the assumption of the current case, Q never receives a removal notification for R_i . It follows from Lemma 32 that Q must receive a $\mathbf{p}_{\text{FLUSH}}\langle f \rangle$ packet from R_i where $f \geq v_m$ before installing view v_m . Assigning $A = B = Q$, $D = R_{i-1}$ and $F = R_i$, we can check that all the conditions of Lemma 37 are met. The only difficulty is with condition 4. There we have

$$r(D) = r(R_{i-1}) \leq v_m \leq f$$

$$R_i \text{ would not send a } \mathbf{p}_{\text{FLUSH}}\langle f \rangle \text{ packet to } Q \text{ if it were aware of its removal, therefore } f < r(Q) = r(B)$$

Therefore process Q receives n .

Case II: $r(Q) > r(R_i)$.

Since process R_i is a member of view v_m , P must receive a $\mathbf{p}_{\text{FLUSH}}\langle f \rangle$ from R_i , where $f = v_m$, before it installs view v_m . Assigning $A = P$, $B = Q$, $D = R_{i-1}$ and $F = R_i$, we can check that all the conditions of Lemma 37 are met. Again the difficulty is with condition 4. There we have

$$r(D) = r(R_{i-1}) \leq v_m = f$$

$$f < r(R_i) < r(Q) = r(B)$$

Therefore process Q receives n in this case as well.

We now turn to the case $i = 0$.

$i = 0$ means that process S , the originator of message n , is a member of view v_m . Therefore S must send a $\mathbf{p}_{\text{FLUSH}}\langle v_m \rangle$ packet to P . Since Q is a member of view v_m , this flush packet must be sent while Q is in $\text{LiveSet}(S)$.

When S broadcasts n , $\text{cur_view}(S) = v_n$ and $\text{v_gap}(S) = 0$, because message broadcasts do not occur when $\text{v_gap} > 0$. As a result the broadcast occurs before S sends the $\mathbf{p}_{\text{FLUSH}}\langle v_m \rangle$ packet to P , and while Q is in $\text{LiveSet}(S)$. This implies that S sends a $\mathbf{p}_{\text{MSG}}\langle n \rangle$ packet to Q as part of the broadcast, and it includes Q in the instability set of n as it adds its record to $\text{WaitSet}(S)$. The message n must stabilize with respect to Q before S sends the $\mathbf{p}_{\text{FLUSH}}\langle v_m \rangle$ packet. This flush packet is sent before S receives a removal notification for Q . Therefore the stabilization can only occur as a result of the receipt of a $\mathbf{p}_{\text{ACK}}\langle n \rangle$ packet from Q . In other words, Q must receive n .

This concludes the proof of the case $i = 0$ and of the Central Lemma. \square

References

- [1] Yair Amir, Danny Dolev, Shlomo Kramer, and Dahlia Malki. Transis: A communication subsystem for high availability. In *22nd IEEE fault-tolerant computing symposium*, pages 76–84, 1992.
- [2] Dan Arnon and Navindra Sharma. An analysis of a group membership service. In preparation.
- [3] Ken Birman, Dahlia Malkhi, and Robbert Van Renesse. Virtually synchronous methodology for dynamic service replication. Technical Report MSR-TR-2010-151, Microsoft Research, 2010.
- [4] Kenneth Birman and Thomas Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pages 123–138, 1987.
- [5] Kenneth Birman, Silvano Maffei, and Robbert van Renesse. Horus: A flexible group communication system. *Communications of the ACM*, 39(4):76–83, 1996.
- [6] Kenneth Birman, André Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems (TOCS)*, 9(3):272–314, 1991.
- [7] Kenneth P Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):37–53, 1993.
- [8] David R Cheriton and Willy Zwaenepoel. Distributed process groups in the v kernel. *ACM Transactions on Computer Systems (TOCS)*, 3(2):77–107, 1985.
- [9] Gregory V Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: a comprehensive study. *ACM Computing Surveys (CSUR)*, 33(4):427–469, 2001.
- [10] Michael Fischer, Nancy Lynch, and Michael Patterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.
- [11] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.
- [12] Leslie Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.
- [13] Nancy Lynch and Alex A Shvartsman. Rambo: A reconfigurable atomic memory service for dynamic networks. In *Distributed Computing*, pages 173–190. Springer, 2002.
- [14] André Schiper and Alain Sandoz. Primary partition virtually synchronous communication is harder than consensus. In *Proceedings of the 8th International Workshop on Distributed Algorithms*, pages 39–52. Springer, 1994.